

AD-A274 128



AFIT/GCS/ENG93D-24

①

S **DTIC**
ELECTE
DEC 23 1993
A

**A METHOD FOR POPULATING THE
KNOWLEDGE BASE OF AFIT'S DOMAIN-
ORIENTED APPLICATION COMPOSITION SYSTEM**

THESIS

**Russell Mark Warner
Captain, USAF**

AFIT/GCS/ENG93D-24

Approved for public release; distribution unlimited

93 12 22 08 9

13090 **93-30965**

**A METHOD FOR POPULATING THE
KNOWLEDGE BASE OF AFIT'S DOMAIN-
ORIENTED APPLICATION COMPOSITION SYSTEM**

THESIS

**Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology**

Air University

**In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering**

**Russell Mark Warner, B.S. in Computer Science
Captain, USAF**

December, 1993

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Availability or Special
A-1	

DTIC QUALITY INSPECTED 3

Approved for public release; distribution unlimited

Acknowledgements

I'd like to thank my advisor, Major David Luginbuhl, for all his help during this thesis effort. Thanks to Major Paul Bailor, who's vision guided the Knowledge Base Software Engineering Group in its efforts, and Dr. Eugene Santos for being on my committee. I'd also like to thank my classmates in the KBSE research group for all their help and support, especially Raleigh Sandy for getting me started and Jay Cossentine for answering (or finding the answer) to my millions of questions.

A special thanks to my wife, Marilee, for her support, patience, proof-reading abilities, and for putting up with a missing-in-action husband for a year. Without her support I could not have completed this thesis, which I dedicate to her. I'd also like to thank the Lord for all his many blessings.

Russell Mark Warner

Table of Contents

	Page
Acknowledgements	ii
List of Figures	vii
Abstract	ix
I. Introduction	1-1
1.1 Background	1-1
1.2 Problem Description	1-4
1.3 Scope	1-5
1.4 Approach	1-6
II. Literature Review	2-1
2.1 Introduction	2-1
2.2 Domain Analysis	2-2
2.3 Systematic Reuse	2-7
2.4 Application Composition Systems	2-9
2.5 Summary	2-9
III. Knowledge Base Population Methodology	3-1
3.1 Introduction	3-1
3.2 Generic Domain-Oriented Application Composition System	3-1
3.2.1 Compose Applications	3-4
3.2.2 Knowledge Base	3-4
3.2.3 Populate Knowledge Base	3-5
3.3 Domain Models and Reuse Infrastructures	3-6
3.3.1 Domain Model	3-8

	Page
3.3.2 Reuse Infrastructure	3-9
3.4 Domain Analysis Research	3-9
3.4.1 Prieto-Díaz's Research	3-10
3.4.2 Arango's Research	3-11
3.5 Knowledge Base Population Process	3-13
3.5.1 Create/Evolve Domain Model	3-14
3.5.2 Abstract Component Behavior	3-15
3.5.3 Design Reuse Infrastructure	3-17
3.5.4 Implement Reusable Components	3-18
3.5.5 Evaluate Domain Development	3-19
3.5.6 Reusable Applications	3-20
3.6 General Process Support and Constraints	3-21
3.7 Summary	3-22
 IV. Instantiation of the Generic Knowledge Base Population Methodology for Architect	 4-1
4.1 Introduction	4-1
4.2 Architect	4-1
4.2.1 The Object-Connection-Update (OCU) Model . . .	4-3
4.2.2 OCU Implementation in Architect	4-4
4.2.3 Creating an Application	4-7
4.3 Technology Base Population Methodology for Architect . . .	4-9
4.3.1 Domain Analysis	4-9
4.3.2 Domain Implementation in Architect	4-11
4.3.3 Evaluate Domain Development	4-20
4.3.4 Reusable Applications	4-23
4.4 Summary	4-23

	Page
V. Implementing the Digital Signal Processing (DSP) Domain in Architect	5-1
5.1 Introduction	5-1
5.2 The DSP domain	5-1
5.3 Domain Analysis	5-2
5.3.1 The Independence Between Domain Analysis and a Domain Implementation	5-2
5.3.2 The Domain Analysis Method Used	5-3
5.3.3 Analyzing the DSP Domain	5-5
5.4 Domain Implementation in Architect	5-13
5.4.1 Implement Domain Model	5-14
5.4.2 Implement Primitive Classes	5-20
5.5 Evaluate Domain Development	5-22
5.5.1 Evolving the DSP Domain Hierarchy	5-23
5.5.2 Testing Primitives	5-25
5.6 Architect Problems that Affect the Use and Implementation of the DSP Domain	5-27
5.6.1 "State" Attributes	5-28
5.6.2 Subsystem Update Algorithm	5-28
5.6.3 Export Values	5-29
5.6.4 Coefficients and Constants	5-30
5.7 Summary	5-31
VI. Conclusions and Recommendations	6-1
6.1 Results of This Research	6-1
6.2 Conclusions	6-1
6.3 Recommendations for Further Research	6-3
6.4 Concluding Remarks	6-6
Appendix A. Domain Information Needed for Architect	A-1

	Page
Appendix B. Digital Signal Processing Examples in Architect	B-1
Appendix C. File Conventions for Architect	C-1
Appendix D. Future Recommended Features for Architect	D-1
Appendix E. REFINE Code Listings for Architect	E-1
Bibliography	BIB-1
Vita	VITA-1

List of Figures

Figure	Page
1.1. A Formalized Software Development Method: Application Composition	1-2
1.2. General System Overview of Architect	1-4
2.1. Domain Analysis Approach Proposed by Prieto-Díaz.	2-3
2.2. Domain Analysis and Software Development.	2-7
2.3. An OCU Subsystem	2-8
3.1. Generic Domain-Oriented Application Composition System (G-DOACS)	3-3
3.2. Producing Reusable Workproducts.	3-10
3.3. Develop Reuse Infrastructure.	3-12
3.4. Knowledge Base Population	3-14
4.1. General System Overview of Architect	4-2
4.2. OCU Subsystem	4-3
4.3. Architect Implementation of the OCU Model	4-5
4.4. Primitive Data-Flow Diagram	4-6
4.5. The Architect Control Panel	4-7
4.6. XOR Primitives and Circuits Domain Technology Base	4-8
4.7. XOR Internal Connections	4-9
4.8. Technology Base Population Process	4-10
4.9. Architect's domain class code for the Digital Logic domain	4-13
4.10. Digital Logic Class Structure	4-14
4.11. Attribute code for the And-Gate primitive class	4-15
4.12. And-Gate portion of the DSL	4-16
4.13. And-Gate portion of the VSL spec	4-17
4.14. And-Gate primitive class icon object definitions	4-17

Figure	Page
4.15. And-Gate primitive class code	4-19
4.16. IconEdit tool showing the And-Gate icon	4-20
5.1. Khoros DSP domain model hierarchy	5-9
5.2. PCDSP domain model hierarchy	5-9
5.3. MatLab DSP domain model hierarchy	5-10
5.4. Architect DSP domain model hierarchy	5-10
5.5. Common filter component visualizations	5-14
5.6. Architect's domain class code for the DSP domain	5-15
5.7. User-defined types and functions for the DSP domain	5-16
5.8. Attribute code for the DFT primitive class	5-17
5.9. DFT portion of the DSL	5-17
5.10. DFT portion of the VSL spec	5-18
5.11. Attribute code for the Sinusoid primitive class	5-18
5.12. Sinusoid portion of the DSL	5-19
5.13. Sinusoid portion of the VSL spec	5-19
5.14. DFT primitive class icon object definitions	5-20
5.15. DFT primitive class code	5-21
5.16. The DSP Technology Base Window	5-23
5.17. Revised Architect DSP domain hierarchy	5-24
5.18. The Test-Primitive Function	5-26
5.19. An accumulator	5-30
B.1. A Four-Sum Moving Average	B-1
B.2. The Input to the Four-Sum Moving Average	B-2
B.3. The Output from the Four-Sum Moving Average	B-3
B.4. The Window-Demo Application	B-5
B.5. The Fourier Transform without Windowing	B-6
B.6. The Fourier Transform with Windowing	B-7

Abstract

This research developed a formal method for adding new domains to Architect, a domain-oriented application composition system being developed at the Air Force Institute of Technology (AFIT) to explore new software engineering technologies. Using canonical formal specifications of domain objects, Architect rapidly composes these specifications into a software application and executes a prototype of that application as a means to demonstrate its correctness before any programming language specific code is generated. Architect is implemented in the Software Refinery environment, which allows Architect to create and manipulate object-oriented specifications. As a part of this research effort, domain-oriented application composition systems were investigated in general, leading to the development of a general method for populating the knowledge base of systems of this type. This general population method was then used as a basis for creating a specific knowledge base population method for Architect. To validate this method, Architect was populated with the Digital Signal Processing domain. The correct implementation of this domain was verified by creating applications and comparing their execution to expected results. The addition of the Digital Signal Processing domain to Architect also serves to validate the usefulness and correctness of the Architect system.

A METHOD FOR POPULATING THE KNOWLEDGE BASE OF AFIT'S DOMAIN- ORIENTED APPLICATION COMPOSITION SYSTEM

I. Introduction

1.1 Background

As software development has increased in both amount and complexity, problems with current software development methodologies have become apparent. Currently, most software is developed in a non-formalized and haphazard fashion—often, a different method is used for each software development effort. The software developed is not portable—current processes are designed to get individual programs out the door, not to generalize or standardize groups of programs. Reuse is not widespread—often the development of each program is treated as a totally new problem without consideration of work done on past programs. There is a semantic gap between software developers and the users that causes problems in communicating requirements from the users to the developers, resulting in software with errors and missing requirements. This gap occurs because software development is done by programmers who are trained in programming methods, but are not necessarily familiar with the area in which that the software is being written. On the other hand, the experts in these areas are often not familiar with good software engineering practices. Also, it is an onerous task to find and correct errors in software developed in this fashion, especially large bodies of code, making verification and validation extremely difficult.

Because of these and other problems, the software engineering community is currently researching methods to improve software engineering, including formal methods (such as formal transformation and verification theories), decomposition methods (object-oriented, etc.), artificial intelligence techniques, automated tools that incorporate software development knowledge, and formalized reuse at both the code and specification level.

The application composition method possesses many advantages over the traditional software development process. Analyzing and storing domain information in this manner is significant because it only needs to be done once and is accomplished by an expert in the domain; some current software development methods also include a form of this analysis, but it is only done at the application level and is usually redone from scratch for every development effort. Also, in current methods this analysis is accomplished by a software expert, not a domain expert who is more knowledgeable in the domain under consideration. Once the domain is established, the reusability provided by the knowledge base decreases development time dramatically since all the domain information needed is already in the system. This domain analysis portion of the application composition methodology also provides formalization and standardization across all applications in the domain, enhancing application development, maintenance, and communication between application designers.

The application composition method provides many other advantages. Validation becomes easier and more complete because formal validation techniques can be performed on the application as it is being composed, instead of waiting until the final product is available for testing. This early validation reduces errors and required maintenance. Similarly, prototyping capabilities and costing analyses are enhanced by the application simulation capability because a "working" system can be created without the effort of developing an executable program. Maintenance time is reduced because modifying a specification (which is at a higher level of abstraction) is less complicated than changing the low-level language code; additionally, because less maintenance is needed and the maintenance is done at a higher level, errors normally created during the maintenance phase are reduced. Also, since the intended user of the new software can perform the function of the Application Designer, the requirements are more likely to be met.

There are many obvious advantages to this new formal software development method; however, its use is still limited because many of the technologies and methods necessary to implement this approach have not been fully developed. Additional research and testing must be accomplished before this method can be implemented on a wide-spread basis.

1.2 Problem Description

The Knowledge-Base Software Engineering (KBSE) group at AFIT has developed a prototype application composition system, called Architect, to research and test this new formal software development methodology. Architect is a domain-oriented software application composer that implements a formal object-oriented software development method with emphasis on reusability. A simplified, high-level conceptual diagram of this system is shown in Figure 1.2. In this figure, the square boxes are actual Architect components while the rounded boxes are processes. In Architect, the knowledge base (described as part of the application composition method) is called the Technology Base.

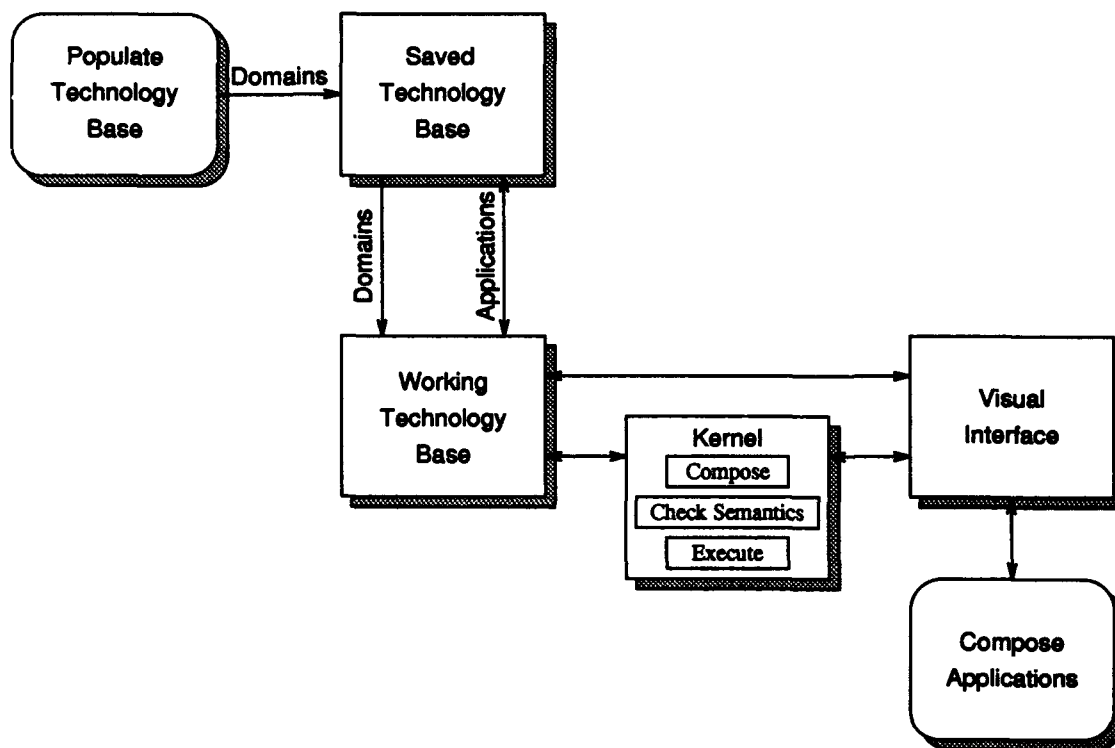


Figure 1.2 General System Overview of Architect

Problem Statement: Investigate and implement a method to populate the Technology Base (knowledge base) in Architect and demonstrate that a more substantial domain can be implemented in this system.

Currently, there is no formalized method for entering domain knowledge into Architect's Technology Base (the Populate Technology Base process). This domain knowledge includes the domain primitives and a domain model, as well as a domain-specific language (DSL) and primitive composition information. Also, restrictions on the domain analysis approach for developing this domain knowledge have not been formalized. Additionally, before this research effort, only two relatively simple domains had been implemented (a pedagogical domain called "widgets" and the digital logic domain). The purpose of our research was to identify and describe restrictions for methods to gather the domain knowledge (domain analysis) and develop a formal method to insert this domain information into the Architect's Technology Base (domain implementation) so that it can be used to create applications. Also, while validating this method, we demonstrated that a more substantial domain can be implemented in Architect so that the system can be used to compose applications in this more complex domain.

1.3 Scope

This thesis effort concentrated only on the Populate Technology Base process for Architect--there were several other concurrent research efforts involving Architect that intersected with this one. These efforts involved improving the visual interface /citeJay, deriving a domain architecture model (12), implementing the Saved Technology Base in an object-oriented database (7), adding an application executive capability (45), and implementing an event-driven domain (41).

Because each of these efforts involved independent modifications to Architect, it would have been intractable to use the most current version as a basis for our research. Thus our research (as well as the others) was accomplished using Architect as it existed (with some minor changes) at the beginning of this thesis effort (nonetheless, every effort was made to ensure that all our work would be compatible). The one exception is that we did incorporate the changes made as a result of Cossentine's research (as well as the changes resulting from this research effort). Because of this independence, the knowledge base population method developed in our research does not include some of the changes incorporated in these concurrent research efforts. For example, this research effort was

not integrated with the database implementation of the Saved Technology Base (implemented by Cecil and Fullenkamp); however, this change from files to a database only affects the form of the Populate Technology Base process, not the functionality. It should be a simple transformation to take the results of this thesis and apply them to the database implementation of the Saved Technology Base. Also, the time- and event-driven execution capabilities added by Welgan requires additional timing information be added to the domain; collecting and implementing this information will need to be added to our method at a future time, but this should be a minor change.

Due to time restrictions, only one domain was implemented as a part of our research. Therefore, we validated our Technology Base Population method using only this implementation. Although every effort was made to generalize, some minor changes may need to be made to this method as other domains with different features are implemented.

This research included some minor changes to the capabilities of Architect itself. Because the implemented domain, Digital Signal Processing (DSP), was more complex than the previously implemented domains, some extensions to the already implemented capabilities need to be made. For example, before this work Architect required that communication between primitives consist only of passing simple data types; the DSP domain required that more complex types be added (see Chapter V). These changes were implemented to make it possible to implement and use the DSP domain.

Finally, the first portion of this research (described in Chapters II and III) were developed jointly with Raleigh Sandy. His work involved populating another application composition system called Automatic Programming Technologies for Avionics Software (APTAS).

1.4 Approach

The following approach was used to reach the objectives (outlined in the problem statement) of this thesis:

- Review current literature. The results of this review are presented in Chapter II. While we found much information on domain analysis, we found little information on specific domain implementation methods and techniques.
- Develop a generic knowledge base population methodology. Part of this effort required us to develop a description of a generic application composition system to define the framework for the generic population methodology. The first few sections of Chapter III present this generic application composition system, while the later sections expound upon our generic knowledge base population methodology.
- Instantiate the developed generic knowledge base population methodology for Architect. Because the methodology developed in Chapter III is generic, it must be instantiated for a particular application composition system before it can be used. Chapter IV shows our instantiation of this methodology for Architect.
- Implement a domain. To validate the population method instantiated in Chapter IV, we used it to populate Architect with the Digital Signal Processing (DSP) domain. This domain implementation, along with developing some DSP applications, also met the objective of demonstrating that a more substantial domain could be implemented and used in Architect. The results of this implementation and how the domain was validated are discussed in Chapter V.
- Identify recommendations and conclusions. Several recommendations for improving Architect and for further research were identified during our thesis effort; the more important ones are discussed in Chapter VI. Also, our general conclusions drawn from this research effort are discussed in this chapter.

This thesis also includes several appendices. Appendix A specifies in detail the domain knowledge required to implement a domain in Architect. Appendix B presents some example DSP applications. Appendix C summarizes the conventions used for the Technology Base files in Architect. Appendix D list the features identified during this research for possible incorporation into future versions of Architect. Finally, Appendix E explains how to get copies of the code created as a part of this research.

II. Literature Review¹

2.1 Introduction

Many researchers are studying methods to encapsulate knowledge needed for software engineering into reusable models. They have proposed ideas to improve knowledge-based software engineering and shorten the gap between software and hardware system development. Some of this research is directly related to our knowledge base population problem.

The technology involved in the effective modeling of application domains is very important to the success of knowledge-based software engineering. Software engineers must develop formal knowledge acquisition processes that solve the knowledge base population problem. Section 2.2 reviews ideas we found useful in our research from current literature in the areas of domain analysis and domain modeling. Both domain analysis and domain modeling focus on the effective modeling of application domains. "There is a strong relationship between [domain analysis] and knowledge acquisition. Building a knowledge base and defining heuristics for an expert system are basically the same problems as [domain analysis]" (13).

Systematic reuse is another topic that supports knowledge-based software engineering. Arango defines systematic reuse as an activity "in which information is systematically acquired and reused in software construction under the control of some management guidelines and costing models" (4:84). The knowledge base must support the code generation component of a knowledge-based software engineering system by providing a library of reusable software specifications or implementations. Section 2.3 reviews ideas we examined in the area of systematic reuse.

There is a class of knowledge-based software engineering systems known as application composition systems that employ techniques like automatic code generation and formal methods to transform specifications into executable code. Because both Architect and APTAS fall into this class of systems, we studied the characteristics of application composition systems. Our study focused on the application composition process and the

¹This chapter was co-written with Raleigh Sandy and also appears in (35).

knowledge base structures. Section 2.4 describes the composition process and knowledge base structures of several application composition systems.

Several terms require definition before we begin our review of the related research. We have already used the term **application domain** that we define as "a coherent set of systems that exhibits common features and functionality across existing and proposed instances" (28). Information occurring within the scope of an application domain, such as functional behaviors and parameters, is considered **domain knowledge**. The term **domain analysis** was first introduced by Neighbors as "the activity of identifying objects and operations of a class of similar systems in a particular problem domain" (26). Prieto-Díaz later defined domain analysis as the process where "information used in developing software systems is identified, captured, and organized with the purpose of making it reusable" (31:47).

2.2 Domain Analysis

Domain analysis is the first, and probably most important, step in adding new information to a knowledge base. Domain analysis was originally adopted as a process to automate several aspects of software development including specification analysis, verification, and application generation (4:82). Early research into domain analysis uncovered the importance of organizing domain knowledge into reusable components. Researchers learned that identifying specific knowledge to reuse through domain analysis was no easy task. Neighbors discovered that "the key to reusable software is captured in domain analysis in that it stresses the reusability of analysis and design, not code" (27) and later proposed a domain analysis method called DRACO. Other researchers have also proposed domain analysis approaches, and we summarize some of these in the following paragraphs.

Prieto-Díaz (32) proposed the data flow model shown in Figure 2.1 that represents his domain analysis approach. In his model, the domain expert (a knowledgeable person in that particular field) and domain analyst (a person with training and experience in analyzing domains) identify and select the domain knowledge. Possible sources for domain knowledge include expert advice, customer surveys, technical literature, and existing implementations, as well as current and future requirements. The domain analyst then assists the domain

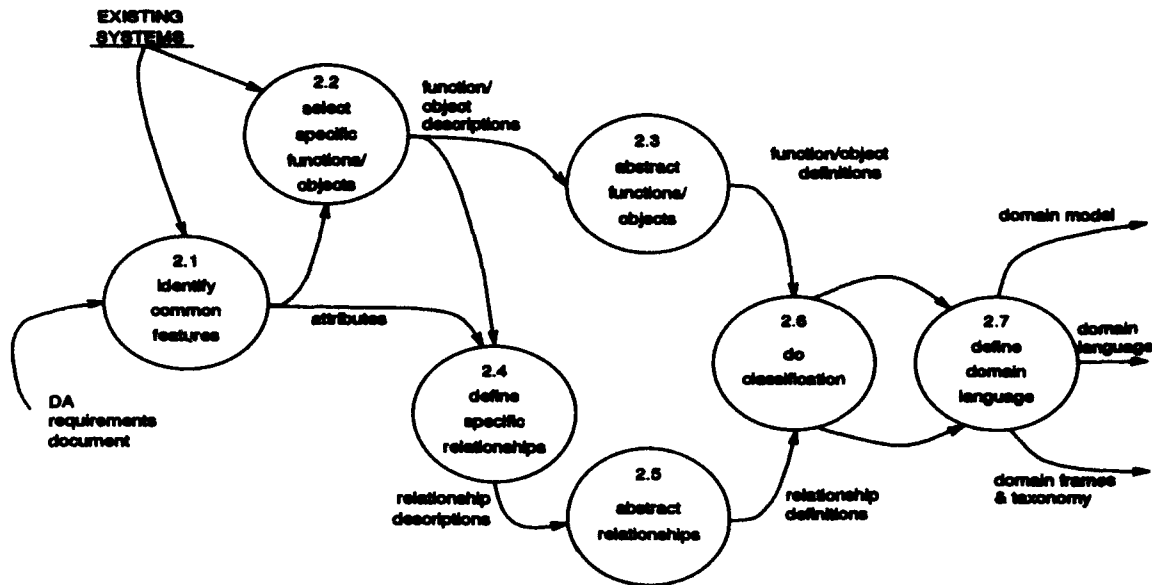


Figure 2.1 Domain Analysis Approach Proposed by Prieto-Díaz (32:67).

expert in abstracting and encapsulating the collected domain knowledge into a subset of the expected outputs (i.e., domain model, domain language, domain taxonomy), as well as domain standards and reusable components. The entire approach is implicitly iterative.

Arango based his view of domain analysis on "the *systematic and incremental approximation* to a definition of an ontology and semantics for a problem domain" (4:83). He proposed an operational definition of domain analysis focused on a reuse-based task (4:83):

Given:

1. a partial, nonformal description of a problem domain
2. a model of a reuser as a learning system

Find: a systematic method to

1. *identify* information in the problem domain which, if available to the reuser in appropriate form, would allow it to attain a specified level of performance,
2. *capture* the information identified as relevant, and
3. *evolve* the acquired information to enhance or maintain the performance of a reuser.

The domain analysis results in a model of the application domain. As with the approach proposed by Prieto-Díaz, this approach identifies and collects reusable domain knowledge.

However, Arango's approach also compares the performance of the reuse-based task to a desired performance level. The domain analysis works to improve the reuse-based task until it reaches the desired performance. Therefore, Arango modeled the reuser as a learning system where improvements to performance correspond to subsequent iterations of domain analysis.

McCain (25) proposed a domain analysis approach consisting of two separate tasks. The first task, application domain analysis, identifies a hierarchy of components and their associations. Application domain analysis is basically the same as other domain analysis approaches studied. This task has three activities: define reusable entities, define reusable abstractions, and perform classification of reusable abstractions. The second task, component domain analysis, defines the individual component behaviors and requirements. This task has four activities to define component interfaces, constraints, algorithms, and customization requirements. McCain's approach is different from other domain analysis approaches by his explicit inclusion of a component domain analysis task.

Kang and others (18) proposed a domain analysis approach called Feature-Oriented Domain Analysis (FODA). The approach identifies prominent features (similarities) and distinctive features (differences) of software systems within an application domain. The features also define mandatory, optional, and alternative characteristics of software systems in the domain. Unlike the other domain analysis approaches we have summarized, the researchers described FODA in sufficient detail to use on large domain analysis projects (ones with several domain analysts). However, this depth of detail can restrict the applicability of the approach.

The Domain Analysis Working Group Report (39) described two domain analysis approaches. The first approach lists the common steps found in other domain analysis approaches:

1. Define Domain Analysis
2. Identify and scope the domain
3. Select a representative set of systems to study
4. Gather inputs for the domain analysis
5. Perform feature analysis at the requirements level

6. Analyze separability, selectability, and trade-offs of features
7. Select an implementation technology
8. Implement and validate products in phases
9. Deliver products of domain analysis

However, they give no real explanation of how a domain analyst accomplishes these steps.

They give more detail for the second approach:

1. Acquire knowledge
2. Perform high-level functional analysis (top-down)
3. Identify objects and operators (bottom-up)
4. Define domain models and architecture

A different view of domain analysis was proposed by Iscoe (14). He focused on the results of the domain analysis rather than a specific approach or the inputs to the domain analysis. He suggested the problem was "to create a model for domain knowledge that is general enough to be instantiated in several domains" (15:299). His approach involved developing levels of "meta-models" that a domain analyst uses to capture the information of a particular application domain. Models consist of objects and their attributes, along with the operations performed upon those objects. This approach had two distinct characteristics: (1) attributes and operations are defined in terms of their underlying scales and (2) object classes use multiple inheritance. Iscoe's approach to domain analysis is known as domain modeling.

Domain modeling is a subset of domain analysis. It is a formal approach to capturing domain knowledge into a specific form that results in a defined knowledge structure called a domain model. We define **domain modeling** as the process of organizing and encapsulating information within an application domain into a predefined knowledge structure. The structure for a domain model depends on the specific application domain being modeled as well as the domain analysis approach applied.

Prieto-Díaz suggested that the structure of domain models "range in level of complexity and expressive power from a simple domain taxonomy to functional models to domain languages" (31:51-52). He defines a domain language as "a collection of rules that relate objects and functions and which can be made explicit and encapsulated in a formal

language and further used as a specification language for the construction of systems in that domain" (32:66). Arango suggested that a domain language documents a "shared paradigm [that] is a precondition for domain analysis" (4:82).

Neighbors developed a domain modeling approach with his DRACO system (27), one of the first systems to specifically employ domain languages and domain models. His approach involved a hierarchy of domains consisting of different levels of abstraction. Domains at the highest level of abstraction are called application domains. Domains at the lowest level of abstraction model conventional programming languages and are called executable domains. Those domains in between are called modeling domains. Application domains span several modeling domains. A domain language defines the external syntax of an application domain. The domain language semantics are written in Backus-Naur form and augmented with control mechanisms.

The domain analysis approaches we have described above are a sample of those approaches published. The software engineer has many options for using or modifying an existing approach. Wartik and Prieto-Díaz (43) presented a strategy for comparing different domain analysis approaches that included the following criteria:

- definition of "domain"
- determination of problems in the domain
- permanence of domain analysis results
- relation to the software development process
- focus of analysis
- paradigm of problem space models
- purpose and nature of domain models
- approach to reuse
- primary product of domain development

Software developers can use these criteria to choose a domain analysis approach that meets their objectives and is within their current resources.

2.3 Systematic Reuse

Wartik and Prieto-Díaz also described three categories of reuse: ad hoc, opportunistic, and systematic (43). Ad hoc reuse is reuse without any formal reuse method. Opportunistic reuse is a software development process with methods to identify the types of reusable components, when to use them, and where they might be found. Systematic reuse is a software development process with methods to define and construct reusable components. They suggested that a software development process could not realize systematic reuse without including the role of domain analysis.

The success of knowledge-based software engineering systems, such as application composition systems, depends on the practice of systematic reuse. Prieto-Díaz suggested that domain analysis could realize systematic reuse by successfully "deriving common architectures, generic models or specialized languages that substantially leverage the software development process in a specific problem area" (31:47). He provided an example of how domain analysis might fit into the software development process (shown in Figure 2.2). Prieto-Díaz claimed that this concept could support several methods of software

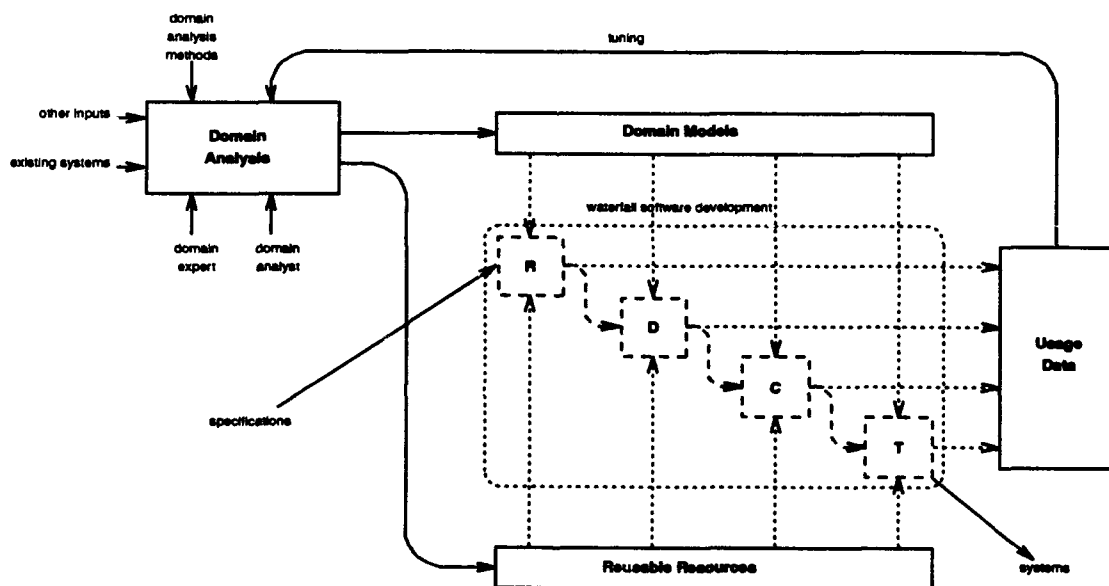


Figure 2.2 Domain Analysis and Software Development (31:52).

development other than the waterfall model. He called this concept a **reuse infrastructure** and stated:

Domain models, in a variety of forms, support (i.e., control) the different phases of software development. Reusable resources are selected and integrated in the new system. Reuse data is then collected and feedback to domain analysis for refining the models and for updating the library. As developed systems become existing systems they are also used to refine the reuse infrastructure (31:52).

Neighbor's DRACO system generates software systems from abstract specifications using its hierarchy of domains. A specification begins in an application domain and gets refined by the system through modeling domains until it can be implemented using an execution domain containing reusable components.

Reusable components, like those in DRACO's execution domains, are very important to systematic reuse. The reusable components must be constructed using some consistent structure called a software architecture. Software architectures define a consistent component structure and also define how to compose applications using a domain's components. Researchers at the Software Engineering Institute have studied systematic reuse and have developed a software architecture called the Object Connection Update (OCU) model (20). Figure 2.3 shows a subsystem in the OCU architecture. Applications are composed of at

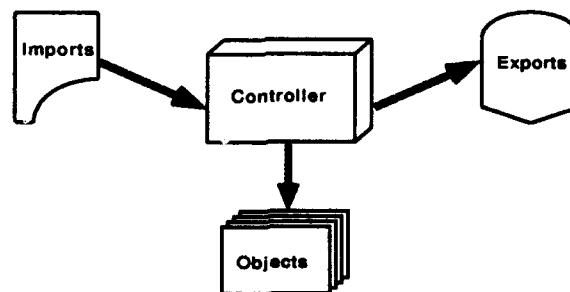


Figure 2.3 An OCU Subsystem (20:18).

least one subsystem under the control of an application executive. Subsystems consist of imports, exports, a controller, and objects. Objects consist of inputs, outputs, and update functions. Gool (12) summarizes the OCU model as well as several other documented software architectures.

2.4 Application Composition Systems

Both domain analysis and systematic reuse play important roles in knowledge-based software engineering systems, especially in the class of application composition systems. There are several application composition systems in use today. Anderson (3), Randour (33), and Weide (44) developed the initial version of Architect, which is an application composition system that implements the OCU software architecture. In this system, domain information is captured in reusable objects at the specification level. Along with these reusable objects is information specifying a domain-specific language (DSL) and visualization specification language (VSL). An Application Specialist (the person creating the software system, called an application) composes applications by either entering a textual specification using the DSL or by visually manipulating icons specified by the VSL. The Architect system is undergoing further study at the AFIT (see research by Gool (12), Cossentine (9), Welgan (45), and Waggoner (41)).

The Lockheed Software Technology Center, under contract with the United States Air Force, prototyped an application composition system (17). This system, called APTAS, "automatically synthesizes executable code from high-level tracking system specifications" (17:1). APTAS generates applications through the support of its Tracking Taxonomy and Coding Knowledge Base. The system uses a software architecture enforced by the knowledge base structure; however, there is no method defined to store information (on existing or new domains) into its knowledge base (see research by Sandy (35)).

Several other application composition systems exist. Some of these include the Kestrel Interactive Development System (KIDS) developed at Kestral Institute (36), the Khoros system developed at the University of New Mexico (40), and the Intelligent Design Aid (IDeA system) developed at the University of Illinois (23).

2.5 Summary

Software reuse has come a long way from ad hoc reuse of low-level code. The reuse of high-level abstractions capturing domain knowledge has become a reality through technologies like domain analysis and domain modeling. Software architectures, combined

with domain analysis, have made it possible for researchers to build software development systems that practice systematic reuse of both low-level code and high-level abstractions. Application composition systems, like Architect and APTAS, have shown that the users themselves can develop their own software systems in a familiar language and environment. Ongoing research in domain analysis and systematic reuse will provide more insight in the development of more operational application composition systems and modeling more application domains. These advances promise to improve the software development process drastically.

III. Knowledge Base Population Methodology¹

3.1 Introduction

Many researchers have envisioned software development evolving from the art of hand-writing code to the engineering discipline of combining and specializing reusable components. One such researcher, Michael Lowry (22:630), envisioned Knowledge-Based Software Engineering environments that automate software reuse using domain knowledge captured through domain analysis. The KBSE research group at AFIT is developing formal methods to implement the automated reuse that Lowry envisioned in the above paragraph. The group's work is based on several of Lowry's premises. This thesis is part of the group's work, and it is focused primarily on how to capture the reusable components that Lowry described into an automated software development system (i.e., how to populate the system's knowledge base).

This chapter proposes a general process that can be tailored to populate the knowledge bases of a particular class of software development systems. Section 3.2 defines the particular class of software development systems. Section 3.3 presents our view of domain models and their corresponding knowledge base representations. Section 3.4 describes the domain analysis methods that we found helpful in developing our process. We use these methods in Section 3.5, along with our system definition and domain model view, to develop a general process to populate a system's knowledge base. Finally, in Section 3.6, we support the development of a "general" process and define several constraints to its implementation.

3.2 Generic Domain-Oriented Application Composition System

We developed a knowledge base population process for a specific class of software development systems. There are several characteristics that distinguish this class of systems. Each system has a knowledge base and a process to compose applications. The knowledge base stores reusable components. Applications are specified using these components. Users can modify, save, and maintain applications through the composition process. Also

¹This chapter was co-written with Raleigh Sandy and also appears in (35).

through the composition process, users can simulate the execution of application (before code is created), translate them to some external form (i.e., synthesize code), and execute them outside the system. These characteristics, including the ability to synthesize executable code, describe the class of application composition systems. However, our class of systems has a knowledge base that must be organized into application domains in an object-oriented fashion. Therefore, we refer to this class of systems as **Domain-Oriented Application Composition Systems (DOACS)**.

There are several advantages to this type of system. The most important advantage is systematic reuse. Reuse is not limited to the code level, but occurs at all levels, primarily at the specification level. Maintenance also occurs at the specification level instead of at the more difficult code level. The application composition process, with its ability to simulate specification behaviors, provides an ideal environment to develop rapid prototypes. This type of system can also provide the powerful capability of creating systems within a graphical environment. The user works with the components and does not have to possess expert knowledge of the domain (e.g., specific algorithms). A user does not need traditional programming knowledge to create new applications, nor to maintain existing applications, because the system automatically generates applications from their specifications. Users can possibly choose between different hardware platforms and programming languages when generating code. Finally, these systems could automate the "housekeeping" chores (e.g., configuration management) so users can concentrate on the more important task of application specification.

It is important to make the distinction between domain-oriented and domain-specific application composition systems. A domain-specific system can be used to compose applications in only one domain and new domains cannot be added. A domain-oriented system can be used to compose applications in any domain implemented in that system and more domains can be added. While there are similarities between these two types of systems, the fact that domain-specific systems contain only one domain greatly simplifies the creation of the system and, of course, nullifies the problem of knowledge base population since the domain information is integrated into the system when it is built. Although domain-specific systems are limited to one domain, the creators of the system are able to take advantage

of the features of that particular domain when designing the system (e.g., the system can be custom tailored around the architecture that best fits that particular domain). Because of this, domain-specific systems have an advantage over domain-oriented systems in ease of composition and capabilities in that particular domain. However, it is not practical to build domain-specific systems for every application domain. Due to their modularity, it is easier to update domain-oriented systems with new software engineering techniques. Also, only one system has to be updated to take advantage of any new technique for many domains as opposed to updating several domain-specific systems (it would be a significant effort to update each system separately).

Figure 3.1 contains the major characteristics of our generic DOACS (G-DOACS). We

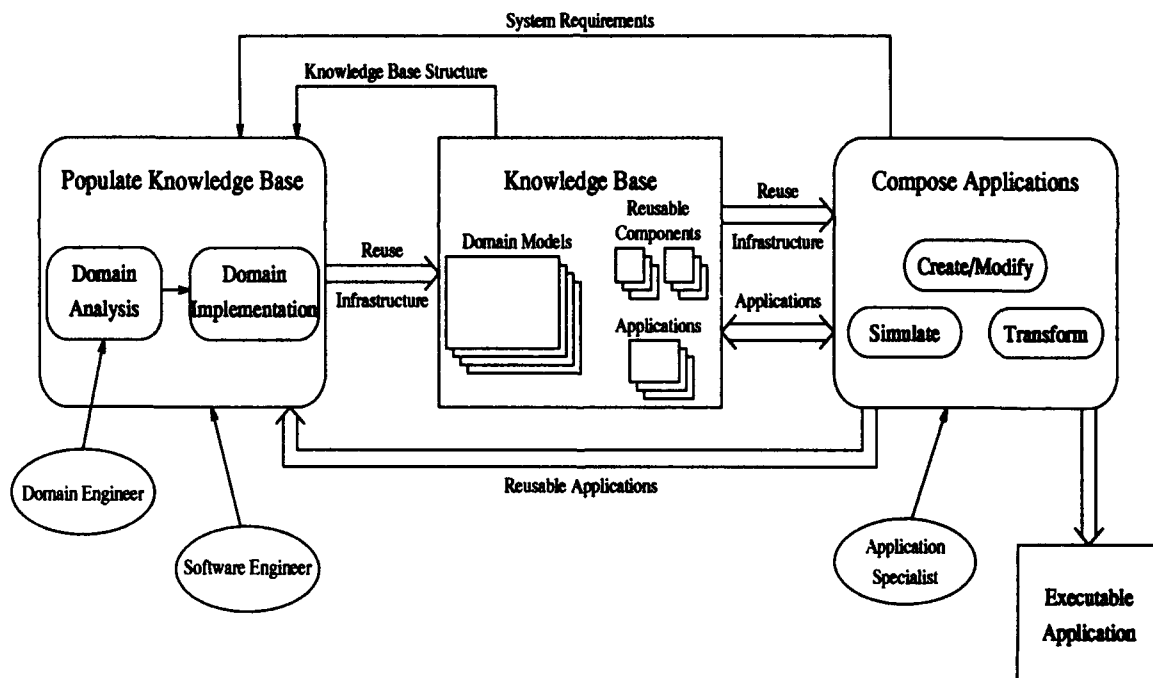


Figure 3.1 Generic Domain-Oriented Application Composition System (G-DOACS)

have used rounded boxes to represent processes and subprocesses (e.g., **Compose Applications** and **Create**), regular boxes to represent physical structures (e.g., **Knowledge Base**), and ovals to represent the roles of the people involved. Notice that we have also included another important characteristic to the G-DOACS definition, the addition of a **Populate Knowledge Base** process that performs the actual knowledge base population.

3.2.1 Compose Applications. An Application Specialist uses the Compose Applications process to create, modify, and validate (through simulation) software applications. The application can then be transformed into executable code. The specific methods to accomplish this process may vary from one DOACS to another.

In general, the Application Specialist creates an application by choosing components from a domain-specific component library, specifies how those components connect together, and declares any necessary processing information. The Application Specialist does not add any new functionality (i.e., no new code), but does specify a component's particular functionality by setting various component attributes. The ability to specify components from several domains within a single application is not a requirement for our class of systems, but this is a desired capability.

Rapid prototyping can be easily accomplished through the simulation capability. The Application Specialist can quickly compose an application and simulate its execution. If the behavior meets the requirements, then the Application Specialist can continue to refine the application; if not, then the Application Specialist can modify the application or throw it away and start over.

We use the term "simulating" rather than "running" because no code has been generated. The system uses the current application specification and any selected reusable components to simulate the behavior that would be expected if code had been generated and executed. Through simulation, the Application Specialist can validate the application's behavior and modify the specification until it meets the desired behavior.

After the application is validated, the Application Specialist can transform it into an executable form (i.e., synthesize code) for a particular target platform. At any stage of application development, the Application Specialist can save the current application specification. This environment also supports application maintenance by allowing the Application Specialist to load and then modify the application.

3.2.2 Knowledge Base. Although specific knowledge bases vary, every DOACS knowledge base contains at least three distinct types of information: applications, reusable components, and domain model representations.

Applications are compositions of the reusable components, along with composition information (e.g., how they are connected, execution order). Therefore, applications contain either links to the reusable components they employ or copies of each reusable component. If links are maintained, then the attribute values that have been changed by the Application Specialist are also saved.

Reusable components (we will often refer to them as just components) are the objects that are connected together to build applications. Reusable components are either primitives or reusable applications. **Primitives** are independent objects that capture the behavior and attributes of objects and classes specified during domain analysis and identified in the domain model. **Reusable applications** are those applications (or parts of applications) identified for potential reuse within future applications. They are somehow processed to make them available to the Application Specialist for composition into applications just like the primitives.

Domain model representations are formal structures that organize the reusable components and other domain-specific information (such as data types, semantic rules, and perhaps even specific architecture information) within an application domain. The types of information in a particular representation depend primarily on the specific application domain and on the chosen approach to domain analysis. We discuss our view of domain models and their representations in more detail in Section 3.3.

3.2.3 Populate Knowledge Base. The **Populate Knowledge Base** process is the focus of our research and is the topic of the rest of this chapter. Briefly, knowledge base population is a process in which the **Domain Engineer** captures domain information as high-level abstractions, and the **Software Engineer** represents these abstractions in a form that is stored directly in a particular system's knowledge base.

In our general process, population begins by selecting an object-oriented **Domain Analysis** approach. The Domain Engineer models a particular application domain using the domain analysis approach chosen. The result of the domain analysis is a domain model and individual component abstractions (i.e., component behavior definitions).

The Software Engineer uses the domain model and the individual component abstractions to create the formal structure of that domain in the knowledge base. We call this construction of a domain model representation the **Domain Implementation**. The domain model representation is a particular instantiation of a domain model and the individual component abstractions for a specific DOACS knowledge base. Once instantiated, the Software Engineer adds the domain model representation to the knowledge base. The Application Specialist can then access any new information when composing applications in that domain.

We borrowed the term **reuse infrastructure** from Arango (4) and Prieto-Díaz (31) to refer to a domain model representation. We developed our process to keep the domain model as independent of the particular DOACS and its knowledge base structure as possible. This independence delays (for as long as possible) the addition of any particular system constraints to the analysis process. The view of a domain model being different from its reuse infrastructure is important to our development of a general knowledge base population process.

3.3 Domain Models and Reuse Infrastructures

The terms domain model and reuse infrastructure are central to our research and, in our opinion, are ill-defined in the current literature where they take on many different meanings. In this section, we define the meanings of these two terms with respect to our research.

Many researchers have viewed the results of a domain analysis as a set of reusable software components and composition rules that capture and implement the semantics of applications within the domain. Given this interpretation, however, Domain Engineers must know the particular knowledge base structure before completing their domain analysis. Domain analysis becomes a task of finding, identifying, organizing, and implementing reusable components. Domain Engineers become the people responsible for populating the knowledge base and collecting the results of their domain analysis within the knowledge base structure itself.

This approach can lead to quick and efficient domain implementations, but can also lead to several problems because of inherent limitations in any knowledge base. Because all domain information cannot be stored in a particular knowledge base, it is difficult to reuse the domain analysis results to populate the knowledge base of another DOACS. Also, problems can occur when identifying or changing the design of the domain model or making other changes as new information is discovered (e.g., better domain implementation methods), because some domain information may have been lost through design decisions when populating the knowledge base. In addition, if the Domain Engineer views a domain through the structure of a particular knowledge base, it will influence the interpretation of domain knowledge and may result in missed or incorrect domain encapsulation. This problem is similar to the difficulty in identifying seemingly simple solutions to a problem when viewing it through the wrong paradigm.

For these reasons, among others, we make a distinction between the results of the domain analysis (domain model and component abstractions) and the implementation of the domain (or reuse infrastructure) in the knowledge base. Therefore, although many researchers have assumed the domain model and its reuse infrastructure (domain model representation) are one and the same, we agree with Arango:

Models of domains and reuse infrastructure should be treated as separate entities, conceptually and practically. Models of domains capture the results of the learning process in domain analysis and support the application of learning techniques. Reuse infrastructures are specified to support the efficient reuse of the information from the model in particular environments (4:88).

This division between domain analysis and domain implementation that we are proposing is similar to a division in compilers. Compiler theory makes a distinction between the intermediate code generated by the front end (analogous to our Domain Analysis) and the machine code generated by the back end (analogous to our Domain Implementation). The front end of the compiler includes those portions of the compiler "that depend primarily on the source language and are largely independent of the target machine ... [while] the back end includes those portions of the compiler that depend on the target machine, and generally, these portions do not depend on the source language, just the intermediate language" (1:20). The front end can be created once for a language, and then different back

ends can be combined with it to create compilers for different machines. In our generic methodology, we propose that domain analysis and domain implementation are analogous to the front and back ends of a compiler. The results of one domain analysis can be used to populate different DOACSSs.

3.3.1 Domain Model. The software engineering community has many different views on what constitutes a domain model. In G-DOACS, a domain model is a structure that captures an application domain's individual components (including their attributes and operations), relationships between components, and other related information (such as shared data types, global operations, composition rules, and architecture information). Prieto-Díaz suggested that the the purest form for a domain model would be a domain language (31:52) that captures all the information about a domain listed above. The syntax of such a language would capture the types of components (with their attributes) and the ways they can be combined, while the semantics would capture the the behaviors of component combinations.

For our process, we chose not to include the individual component behaviors (semantics) as part of the domain model. We separated the component behaviors from the domain model because defining behaviors is often the most difficult task during domain analysis. Also, users can compose applications with a well-defined domain model implementation but with only partially defined (and implemented) component behaviors. This allows a Domain Engineer to quickly capture a small "core" of domain knowledge (the domain model plus a few of the component behaviors) that, once implemented, provides the Application Specialist the capability to compose simple applications before many of the component behaviors have been defined. Under our definition, the domain model contains only the information that fully describes the syntax of an application domain; individual component behaviors are defined and implemented separately.

In this chapter and those that follow, we describe two instances of the domain model. The first is the domain model created during the domain analysis process (as described in the preceding paragraph); the second is the implementation of the domain model in the

knowledge base. The instance we are identifying with the term domain model should be clear from the context.

3.3.2 Reuse Infrastructure. As stated previously, the results of the domain analysis should be independent of any particular DOACS. So ideally the domain analysis outputs a domain model and component abstractions without constraints to their usefulness to any particular DOACS. This approach follows the established software engineering practice of pushing design decisions down to the lowest possible level. If the domain analysis is done correctly, the domain model and component abstractions can be evolved over time without the need to reanalyze the whole domain. The domain analysis results can also be used to populate the knowledge base of any DOACS (i.e., the results are reusable²).

Since the domain analysis results are derived independent of the knowledge base structure, the system cannot use them to generate applications. Therefore, the Software Engineer must organize and implement the domain model and component abstractions into the correct knowledge base representation for some particular DOACS. We call this instantiation of the domain analysis results the reuse infrastructure. The reuse infrastructure implements the information captured in the domain model and component abstractions in the form required by the structure of a particular knowledge base. Traditional software engineering methods apply to the development of any reuse infrastructure.

Separating the reuse infrastructure from the domain analysis results allows us to develop our knowledge base population process without introducing constraints too early in the process. Before presenting our process, we will discuss several theories that were helpful in our research.

3.4 Domain Analysis Research

In Chapter II, we summarized the current literature in the field of domain analysis and discussed the relation between this field and our research in knowledge base population. This section summarizes some of the contributions of two recognized researchers in the

²Methods to capture domain information in an object-oriented database feeding the knowledge bases of several DOACS are addressed by Cecil and Fullenkamp (7).

field: Prieto-Díaz and Arango. Prieto-Díaz (32) proposed a functional model of a domain analysis process while Arango (4) explored the domain analysis process in a formal software reuse system. Our process is built upon many of their contributions.

3.4.1 Prieto-Díaz's Research. According to Prieto-Díaz (32), domain analysis captures the “essential functionality” of components, which assists the application developer. He proposed the data flow diagram in Figure 3.2 with three activities that are involved with domain analysis: prepare domain information, analyze domain, and produce reusable workproducts. These three activities comprise the task of knowledge base population.

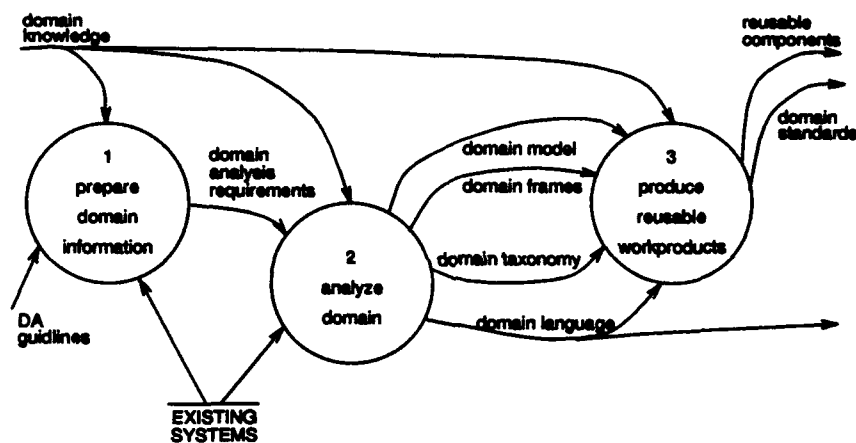


Figure 3.2 Producing Reusable Workproducts (32:67).

The prepare domain information activity produces the requirements of the domain analysis. This activity includes bounding the application domain, identifying the sources for domain knowledge, selecting a specific domain analysis approach, and defining the expected results.

The analyze domain activity uses these requirements to produce collections of domain abstractions including the domain model, domain frames, a domain taxonomy, and a domain language. This activity is the domain analysis theory proposed by Prieto-Díaz – we previously presented the details of the analyze domain activity with the data flow diagram in Figure 2.1. The domain abstractions capture the behavior of objects within

the domain, identify their relationships, and model the structure of these relationships. Prieto-Díaz suggested that the ideal result of domain analysis is the domain language.

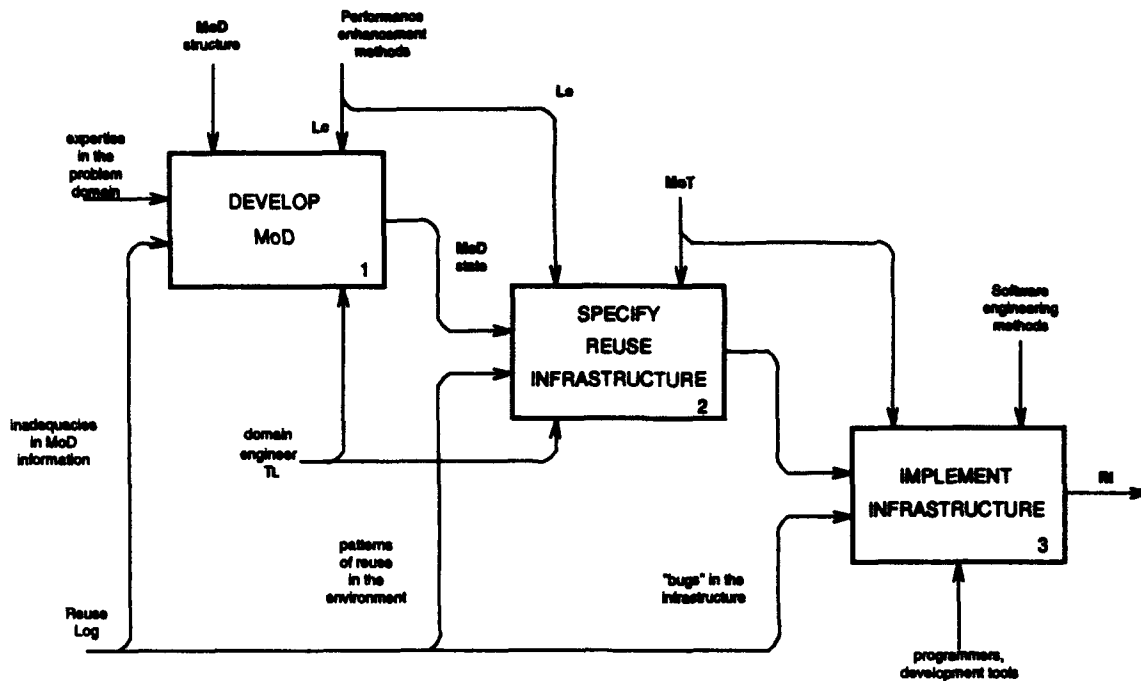
The produce reusable workproducts activity takes the domain abstractions and produces a set of reusable components. The components implement the objects and relationships identified in the domain model and used in the domain language.

Prieto-Díaz proposed a functional model that successfully identifies the relationship between domain analysis and the production of a reusable infrastructure. It also successfully defines several outputs involved in the process; however, it does not sufficiently describe the requirements of each activity. For instance, there is nothing that constrains the structure of the reusable components. Systematic reuse cannot be realized without such constraints. His model does not explicitly capture the importance of feedback and iteration in the domain analysis process, nor does it identify the role of the knowledge base in separating the reuse infrastructure from the domain model.

3.4.2 Arango's Research. Arango (4) outlined a "domain engineering framework" based on the concepts of software reuse. His framework serves as a structure for synthesizing a tailored domain analysis process. Arango suggested his framework has general application because reusers are modeled as learning systems.

Figure 3.3 describes the learning component using boxes from the Structured Analysis and Design Technique. The component consists of three activities and is defined by a set of state variables (4:84):

- **Exp:** expertise in the domain
- **ReuseLog:** feedback from the reuse task
- **RI:** reuse infrastructure
- **TL:** technologies to support the representation and evolution of the domain model
- **MoD:** domain model
- **Lc:** method to increase coverage
- **Le:** method to improve efficiency



- **MoT**: model of the reuse task

The develop MoD activity represents the actual domain analysis and results in a domain model. Arango made a clear distinction between the domain model and the reuse infrastructure. He stated that the distinction is analogous to the distinction between “representations for systems specifications and programming languages for systems implementations” (4:82). This distinction allows the domain model to be independent from the MoT.

The other two activities use the current state of the domain model to organize and implement a reuse infrastructure. The particular organization and implementation depend heavily upon the particular **MoT**. Traditional software development procedures are applicable during these two activities.

Arango's learning component is very similar to the functional model proposed by Prieto-Díaz. It separates the creation of a domain model from the development of the reuse infrastructure. Arango successfully identified the various traits involved in the development

of a reuse infrastructure. He also explicitly indicated the roles of feedback and iteration. However, his state variables were too ambiguous. He did not sufficiently describe the model of the reuse task or the role of the knowledge base, which are essential in developing the reuse infrastructure. There was also no clear distinction between the results of infrastructure specification and those of infrastructure implementation. Prieto-Díaz had combined these two activities into his produce reusable workproducts, but Arango seemed to think there should be some distinction. Our research combines these two processes into a single knowledge base population process. We attempt to clarify all traits involved in knowledge base population without adding additional constraints to the process.

3.5 Knowledge Base Population Process

We propose a five-step process to knowledge base population that explicitly identifies the roles of the Domain and Software Engineers, incorporates feedback, and iteratively captures a domain (in stages). Each step has defined inputs, outputs, methods, and constraints. Our process is shown in Figure 3.4, using boxes from the Structured Analysis and Design Technique. The methods (bottom arrows) driving each activity or step must be selected prior to performing the activity. The constraints (top arrows) show the requirements that drive each activity. The inputs and outputs (left and right arrows, respectively) show the domain information as it is captured and passed from activity to activity.

Activities one and two compose our domain analysis process and result in a domain model and component abstractions. As discussed in previous sections, this domain analysis is independent of the particular DOACS. Activities three and four comprise our domain implementation process that results in a reuse infrastructure and, as can be seen from the constraints, that is very dependent on the particular DOACS. This division between domain analysis and domain implementation is such that one domain analysis can feed multiple domain implementations for different DOACS. Activity five provides an evaluation and feedback mechanism to iteratively improve the domain model, component abstractions, and the reuse infrastructure. The remainder of this section defines each activity and all its associated traits.

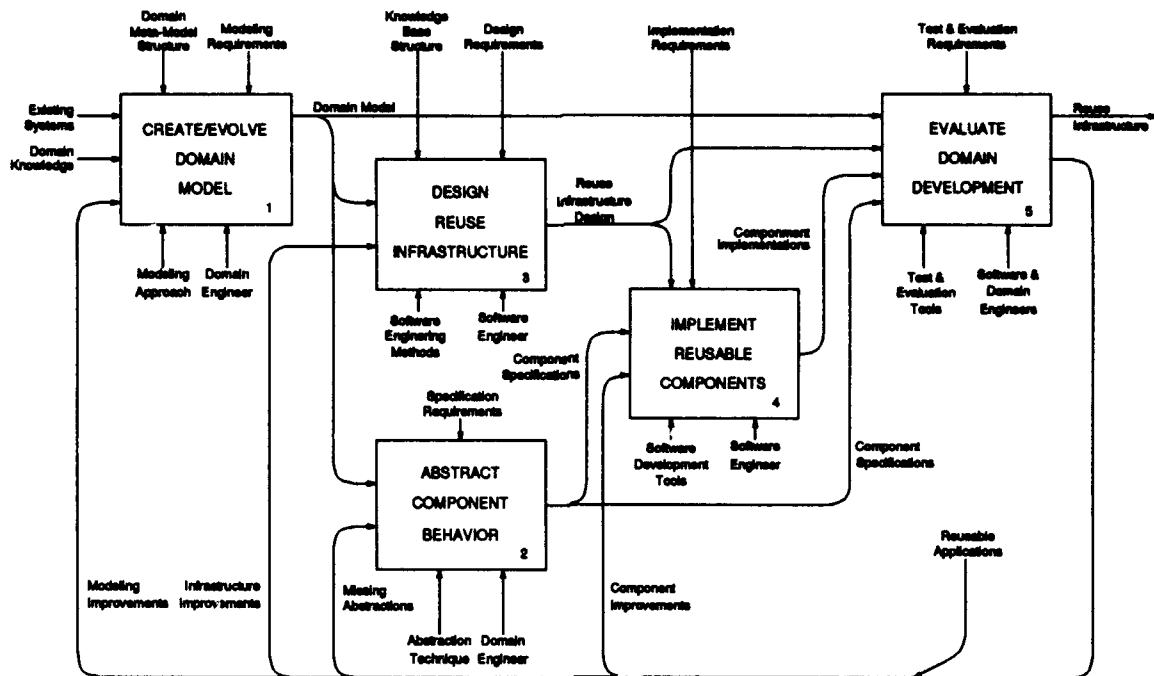


Figure 3.4 Knowledge Base Population

3.5.1 Create/Evolve Domain Model. The first step in populating the knowledge base of a DOACS with a particular domain is generating a domain model. A domain model is created during the initial domain modeling³ activity and periodically updated through subsequent iterations. Before performing the initial iteration of this process, the methods and constraints must be defined, including the specific domain modeling approach and the formal modeling requirements.

The Domain Engineer should select the domain modeling approach based on the strengths and weaknesses of the approach and on the characteristics of the particular domain to be analyzed. We recognize that factors such as the Domain Engineer's familiarity with certain modeling approaches, management's overall goals, and the availability of resources will also play an important role in choosing a specific approach. Any well-defined object-oriented modeling approach should be sufficient since our methodology requires

³In keeping with our definitions presented in Chapter II, we considered domain modeling to be a type of domain analysis that results in a domain model with some formalized structure.

consistency and completeness⁴ of the resulting domain model, but puts no restrictions on the form of this model (other than object-oriented). The Domain Engineer is the one who will have to use the approach to create and update the domain model. Therefore, the approach must be thoroughly understood. Also, the approach should take advantage of any available automated domain analysis/modeling tools to aid in capturing the domain. Such tools could improve performance and could aid in maintaining the domain model's consistency and completeness (information on one such tool can be found in Crowley's research (10)). We expect that these tools will be tied together after they (and DOACS) become more established (effectively automating much of the role of the Software Engineer).

The modeling requirements come from both the domain modeling approach chosen as well as the fact that domain analysis must be accomplished using object-oriented techniques. The minimum modeling requirement consists of a formal structure that encapsulates all the information represented in the domain model. The structure is similar to a meta-model structure (proposed by Iscoe (14)) and should be in some object-oriented form.

Once the domain modeling approach and modeling requirements have been defined, the Domain Engineer can begin creating the domain model. The domain model is transformed from a simple hierarchy of abstraction identifiers to a formal structure of captured domain semantics (which ideally leads to a domain-specific language). As the domain model evolves, the Software Engineer can begin applying the constraints of a knowledge base for a particular DOACS to construct a reuse infrastructure (i.e., instantiate the domain model). However, before discussing the domain implementation, we must address the task of defining individual component abstractions.

3.5.2 Abstract Component Behavior. After a structure has been created containing components in the domain, the behavior of the individual components must be defined. McCain refers to this process as "component domain analysis" (25:73). The abstraction

⁴By completeness, we do not mean that the domain model has to capture the whole domain, but rather that the parts of the domain that it does capture are completely captured within the scope of the domain analysis approach.

technique chosen by the Domain Engineer must be consistent with the modeling approach used in the Create/Evolve Domain Model activity (ideally, they both should be part of an integrated domain analysis approach). The results of the abstraction technique should be in a form that the Software Engineer understands and can implement. For example, it is possible to create *Z* schemas that cannot be implemented on a computer system. The distinct features of the domain under analysis should also play a role in which abstraction technique the Domain Engineer chooses. However, although different abstraction techniques may be used for different domains, using the same technique within a domain is required.

One of the most important aspects of the component abstraction is the interface specification. Although object interfaces could have been partially defined during the previous activity (if not completely defined, depending on the modeling approach chosen), each component abstraction must have interfaces that support consistent relationships between other component abstractions. The Domain Engineer must keep in mind that this whole process is based on an object-oriented methodology and that the final result will be objects that can be connected together. Incorrect or inconsistent component interfaces will cause severe problems when the Software Engineer tries to implement the abstractions and also when the Application Specialist attempts to compose applications.

Another aspect of each component abstraction is the effect of certain component attribute values. Attributes contain data that represents characteristics of an object (20:20). These characteristics may have a large impact on the behavior of a particular component. The Domain Engineer must identify these impacts during this activity. For example, a component that converts a real number to an integer may have an attribute that specifies whether that component will do the conversion by rounding to the nearest integer, truncating, or rounding up to the next integer. One useful feature of a DOACS is the ability for the Application Specialist to change these component attributes and "tailor" the component for a specific application. However, components can have attributes that should not be changed by the Application Specialist (for example, attributes that reflect the internal state of the component). These internal attributes should also be identified during this activity.

It is expected that the Domain Engineer may identify better ways to model the domain at this stage of the process. For example, at this point, the Domain Engineer may choose to combine objects or classes (generalization), or split an object or class (specialization). These improvements are acceptable (and even desirable). As stated before, our process is iterative. Component abstractions are evaluated (by the Evaluate Domain Development activity), and the Domain Engineer is notified of any problems. Our study of domain analysis revealed that the best domain model is the one that has evolved over time (i.e., it is difficult to come up with the best model on the first iteration).

We recognize that the Domain and Software Engineers may choose to merge this activity with the Implement Reusable Components activity by using a DOACS-specific method to define and implement individual component behaviors. This could save time and effort during the domain analysis, but it has its drawbacks. When system requirements and implementation requirements constrain the definition of individual component behaviors, biases could easily enter into the component definitions (e.g., implementation details often "muddy the water" of a "pure" domain analysis). Any bias introduced at this early stage may cause difficulties as the domain implementation evolves. Also, abstracting component behaviors in this way may make it difficult (or impossible) to reuse the abstracted components to populate another DOACS.

3.5.3 Design Reuse Infrastructure. The domain model is a structure (much like the syntax and semantics of a grammar) that captures the individual components, relationships between components, and other related information in an application domain. The Software Engineer must now organize the information captured in the domain model into a form that can be stored in the knowledge base of the particular DOACS. The organized abstractions become the domain's reuse infrastructure design. The infrastructure design is similar to Arango's reuse infrastructure specification that acts as "an architecture for reusable information"(4:82). In designing the reuse infrastructure, current software engineering methods should be used; however, these methods must fit into the object-oriented paradigm. Also, the Software Engineer must understand and follow any other design requirements (e.g., organization-specific design standards).

The implementation of this process is very system-specific; however, the results of this activity will usually form, as a minimum, an abstract syntax tree with each node representing some abstraction and each leaf representing a component implementation. More sophisticated DOACS will require more sophisticated results like implemented domain-specific languages, domain semantics (e.g., a domain rule that component A must follow component B), domain-specific software architectures, and methods for defining how the components are presented to the Application Specialist during the composition process.

3.5.4 Implement Reusable Components. Now that the reuse infrastructure has been designed and the component behaviors have been defined, the individual components must be implemented (transformed into the required "executable" form). Component implementation is a translation of each component from its abstract definition into a form representable in the knowledge base and executable by the DOACS.

The Software Engineer implements each abstraction organized in the domain infrastructure design. The implemented abstractions become part of the reuse infrastructure implementation and, when all of them are done, domain implementation is complete. Before implementing the components, the specific software development tools and the implementation requirements must be identified.

The selection of software development tools will depend on the software implementation languages accepted by the DOACS. The Software Engineer should use whatever tools are available to implement the components. Commercial compilers and software development environments can provide the necessary utilities to perform reuse infrastructure implementation.

The implementation requirements constraint indicates the constraints imposed by the DOACS, most of which result from the requirements of its knowledge base. These constraints must include a formal description of the software architectures supported by the DOACS and the specific procedures for representing component implementations within the knowledge base. The implementation requirements should also identify different methods to model the component abstractions. For example, a stack data type can be implemented as an array or list with various advantages to each of these implementations. Part

of the implementation requirements should detail the procedures for selecting the desired implementation method for different abstractions.

A priority list of the order to implement the components may also be included in the implementation requirements. The Application Specialists may require some components immediately, while other components may not be needed until well in the future. Using priorities, the Software Engineer could implement components in order of their importance to the Application Specialist. This allows the DOACS to support application composition before complete domain implementation is finished.

3.5.5 Evaluate Domain Development. In this activity, the Software and Domain Engineers evaluate the outputs of the previous activities. This activity covers a broader scope than the others because, while the others are focused on one area of domain analysis/implementation, this activity has relevance to the entire knowledge base process from start to finish.

The results of each of the other four activities should be evaluated as they are generated, both individually and along with results from previously completed activities. Also, when all the results are completed, they should again be evaluated to ensure consistency and completeness. This activity should trigger one or more of the previous activities when errors or better ways to model/implement the domain are discovered. Each activity could include their own evaluations as part of their normal execution, but we have made the evaluation a separate activity to provide a way to evaluate the results of all the activities in reference to one another.

Although we have placed this activity at the end of our process, it is involved at each step of the knowledge base population process. As the domain model and knowledge base evolve, this activity becomes more important in maintaining the integrity of the applications they generate.

An input to this activity that is not specifically shown on Figure 3.4, but is worthy of mention, is from the Application Specialist. As the Application Specialist composes applications, several problems may be identified (such as missing components, errors in components, or problems with component interfaces). The Domain Engineer then evaluates

the problems and makes any necessary changes to the domain model or reuse infrastructure. Another input from the Application Specialist is the identification of applications that should be incorporated into the knowledge base as primitive components. These reusable applications are discussed in the next section.

3.5.6 Reusable Applications. Along with the components identified during the domain analysis, the Application Specialist may want to use an existing application (or part of one) as a component in a new application. We call this reused application a **reusable application** (although all applications are reused in the sense that they can be reloaded and modified). When the Application Specialist is choosing components to use in composing an application, these reusable applications should be included in the list of choices. It is possible that a DOACS could implement reusable applications without any special processing; however, in general, we expect that these applications will need to be identified for reuse and somehow "processed" by the Domain and Software Engineers. This processing is completely dependent on the specific DOACS, and there could even be different methods for accomplishing this within the same DOACS.

The creation of a reusable application may be simple or complex, depending on the specific situation and the capabilities of the system. Primitive application creation could consist of stripping off desired data "sources" and "sinks" (components that generate or consume data that we now want to leave off the application) and identifying the interface to the reusable application that remains after these components are removed. If the specific DOACS does not have the capability of storing and using a reusable application like this, then the Software Engineer may have to somehow combine all the behaviors of the components of this application into this new component in the same manner as those identified in the domain analysis. Also, depending on the specific DOACS, it may be required that this new primitive application be added into the reuse infrastructure design.

Along with making the reusable application available as a new component to the Application Specialist, the Domain Engineer should consider adding it to the domain model (in the Create/Evolve Domain Model activity). If the reusable application is added, then its behavior should be captured in the Abstract Component Behavior activity and

consideration given to implementing it as a single component (rather than a collection of components).

3.6 General Process Support and Constraints

There are many reasons for having a general knowledge base population process. One reason is to provide a framework to use when developing tools and utilities that apply to populating many knowledge bases. Is there justification for developing a general process? How "general" is this process? This section is our attempt to support our development of a "general" knowledge base population process.

We have illustrated a possible knowledge base population process. We attempted to clarify the process by identifying the methods and constraints for each activity; they change depending on the specific application domain and the specific DOACS used to develop applications. This versatility, tying each of these traits to the characteristics of the domain or system, adds justification to the general applicability of the process. Although the domain model resulting from the domain analysis is independent of the knowledge base constraints, the model's structure and the domain analysis method used to create it can differ for each domain. This characteristic is captured in the first stage of our process. The same idea applies to building the reuse infrastructure. The infrastructure can be different for each DOACS. We do not suggest that this process will support every application domain or apply to all DOACSs (it has not even been shown that every domain can be modeled using current modeling techniques). However, we do suggest that this process will support many application domains and many systems.

There are some characteristics that must exist in the domain and the system. The system's knowledge base must have the capability to represent an organized collection of reusable software components and the rules for their composition. We described this framework in Section 3.2. The domain must be somewhat established (i.e., some structure must exist for building applications in the domain, either informal or formal). This implies that the domain is mature enough to have existing applications that could provide important information during a domain analysis.

We have developed our process with the goal of future automation. Constraints placed on each activity (such as knowing the particular software architecture) have been included only where necessary. However, software engineers may be concerned about changes made to these constraints while a domain model and knowledge base are evolving. For example, suppose we modify our knowledge base structure after already implementing several domains. What impact does this have on the knowledge base population process? Will we have to respecify and re-implement the domain model? Are we forced to stick to the original constraints? Although a more general knowledge base population process may exist, we feel that our process can handle these problems if the proper attention is given to the definition of each constraint.

3.7 Summary

A formal process for populating a knowledge base results when we apply the outline described in this chapter to a candidate DOACS. The process that we have developed incrementally evolves the domain – both the domain model and the reuse infrastructure in the knowledge base. The process explicitly captures the role of feedback through the evaluation of each activity. When the process is first started for some application domain, most of the effort will involve the Create/Evolve Domain Model and Abstract Component Behavior activities (activities one and two of Figure 3.4). As the process continues, effort will increase in the Design Reuse Infrastructure and Implement Reusable Components activities (activities three and four of Figure 3.4). The attention given to the Evaluating Domain Development activity will depend on the particular requirements for testing (which usually relates the size and complexity of the domain).

Although we have developed a general process, software engineers need to conduct more application-oriented research into knowledge base population to gain more experience and develop tools to assist in evolving domain models and structuring knowledge bases. The DOACS concept has not yet been proven in the software engineering community. Our research efforts and the efforts of other researchers will help to improve these software development technologies.

IV. Instantiation of the Generic Knowledge Base Population Methodology for Architect

4.1 Introduction

In the previous chapter, we presented a generic knowledge base population process (Figure 3.4) that we maintain could be tailored to any DOACS. In this chapter, we instantiate this process for a specific DOACS: Architect¹. However, before presenting our instantiation of the generic population process, we must first describe Architect.

4.2 Architect

Figure 4.1 (reprinted from Chapter I) shows a general overview of the current version of Architect². The rounded boxes are tasks accomplished by the Application Specialist, Domain Engineer, and Software Engineer; the square boxes are components of Architect.

Architect runs within the Software RefineryTM development environment created by Reasoning Systems, centered around a Lisp-based specification language called REFINE. The REFINE language "provides an integrated treatment of set theory, logic, transformation rules, pattern matching, and procedure" (34). Programs in REFINE consist of executable rules and functions that query and modify a powerful object base; in addition, there are many built-in functions to aid in manipulating this object base. The user defines these rules and functions at the specification level, so REFINE supports programming with "executable specifications". Also included in REFINE is a syntax definition system, called DIALECT, which allows users to easily design and implement new languages, through grammar definitions. These grammars can be used to populate the REFINE object base by parsing in text files, an ideal way to implement domain-specific languages.

Architect fits well into the DOACS concept presented in Chapter III. It is a domain-oriented application composer that has a knowledge base, called the Technology Base, to encapsulate domain knowledge. Through the Architect Visual System Interface (AVSI), the Application Specialist can view domain documentation, compose or load an application,

¹In Chapter IV of (35), Capt Sandy instantiates this same process for a DOACS called Automatic Programming Technologies for Avionics Software (APTAS).

²A summary of other research accomplished concurrently with this effort is in Chapter I.

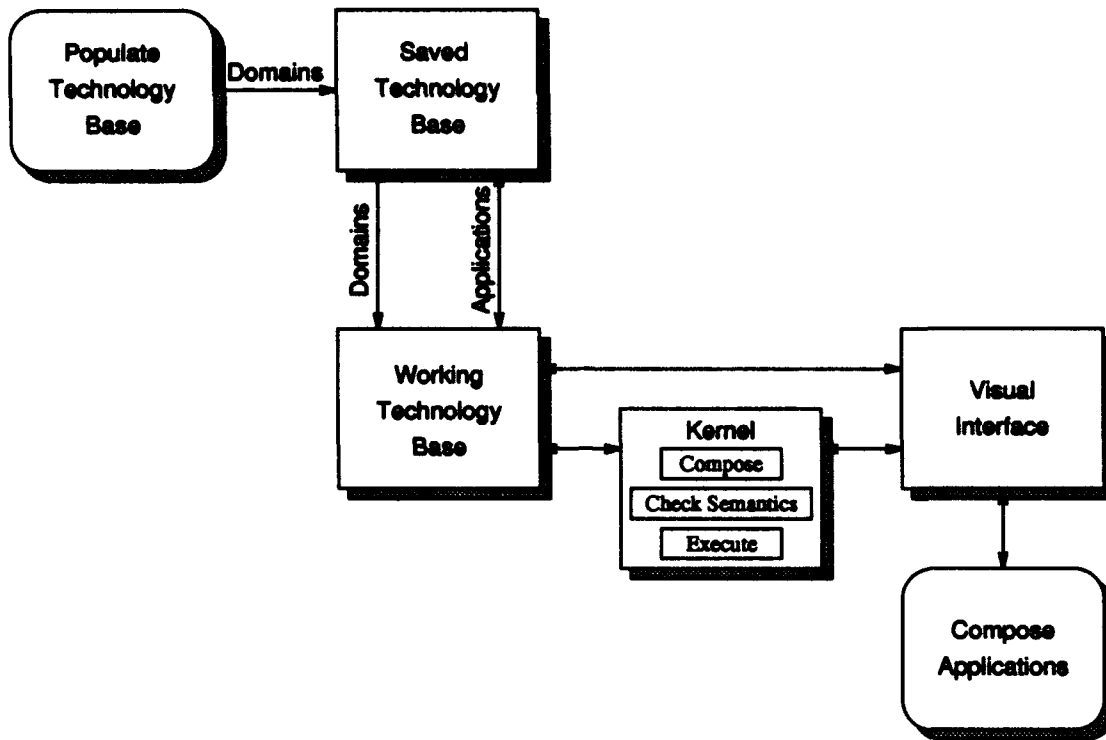


Figure 4.1 General System Overview of Architect

check the semantics of the application, execute the application, and save the application. An application is composed of subsystems, and subsystems are composed of primitives and/or other subsystems. The Working Technology Base is contained in the REFINE object base (non-persistent between REFINE environment sessions); the Saved Technology Base currently consists of a directory structure containing text files³. Architect does not yet have a sophisticated code generation capability (applications can only be “run” within the scope of Architect); however, it is expected that this capability will be added in the future.

Currently, Architect has only one architecture available, which is based on the OCU model. This model is summarized in the following section, followed by a section describing the Architect implementation of this model. Finally, a description of how to create an application is given.

³Information on implementing the Saved Technology Base in a database is contained in (7).

4.2.1 The Object-Connection-Update (OCU) Model. The OCU model (20) was developed by the Software Engineering Institute (SEI) as a part of their Software Architectures Engineering Project. In the OCU model, applications consist of a set of subsystems (see Figure 4.2) that contain:

- A Controller: manages the objects in the subsystem. A subsystem's controller can be accessed through the following procedures: Update, Stabilize, Initialize, Configure, Destroy.
- An Import Area: the collection of the data needed by the objects in the subsystem. It consists of a collection of the input_data of all the objects and imports of all the subsystems that are contained in this subsystem.
- An Export Area: the collection of the data produced by the objects in the subsystem. It consists of a collection of the output_data of all the objects and exports of all the subsystems that are contained in this subsystem.
- Objects and other subsystems: subsystems can be nested to any depth.

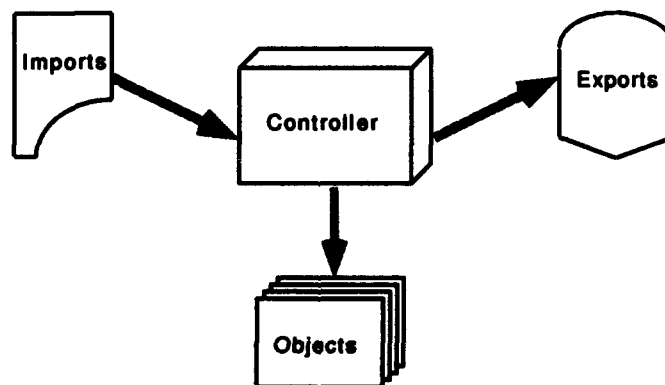


Figure 4.2 OCU Subsystem

“An object models the behavior of a real-world or virtual component and maintains state. All algorithms that are necessary to model the behavior of the component are localized in an object” (20:19). Objects have:

- An Interface: consists of five procedures:

- Update: calculates the new object state data based on the current state and input values.
 - Create: creates a new instance of an object.
 - SetFunction: switches or redefines (by coefficient modifications) the update procedure.
 - SetState: modifies the object's state data directly.
 - Destroy: deallocates the object.
- Input_Data: data provided by other objects used to calculate the new state data.
 - Output_Data: data resulting from an object's state calculations that is externally visible.
 - Attributes: data that represent characteristics of an object.
 - Current_State: data representing the current condition of the object.
 - Coefficients: data used in the algorithms for calculating the object's state data.
 - Constants: data that represent unalterable object attributes.

It is important to note that while the SEI developed the OCU model, to our knowledge they have not developed any automated tool support systems for this model. Because of this, there are some areas of the OCU model that need more detail in order to automate this model. For example, exactly what form an object's State_Data takes is not defined. Therefore, during AFIT's automation of the OCU model, these areas had to be defined to a greater level of detail: these areas have caused some problems in the Architect OCU implementation. Some of these problems are described below and in Chapter V because they affect the implementation of a domain in Architect.

4.2.2 OCU Implementation in Architect. Because there are some areas of the OCU model that are not completely defined to a sufficient level of detail to be automated, and because the integration of the OCU model in any DOACS requires that implementation decisions be made, the OCU model implemented in Architect differs somewhat from the OCU model described above. In this section we identify the important differences.

A graphical representation of the structure of applications is shown in Figure 4.3. In the notation used in this figure⁴, the boxes represent object classes (there may be many objects of these types) and the lines represent associations between objects of the connected classes. Black circles on the associations show that there can be one or more of that type of object on that end of an instance of that association. An object class has three parts: name, attributes, and operations, which are divided by lines.

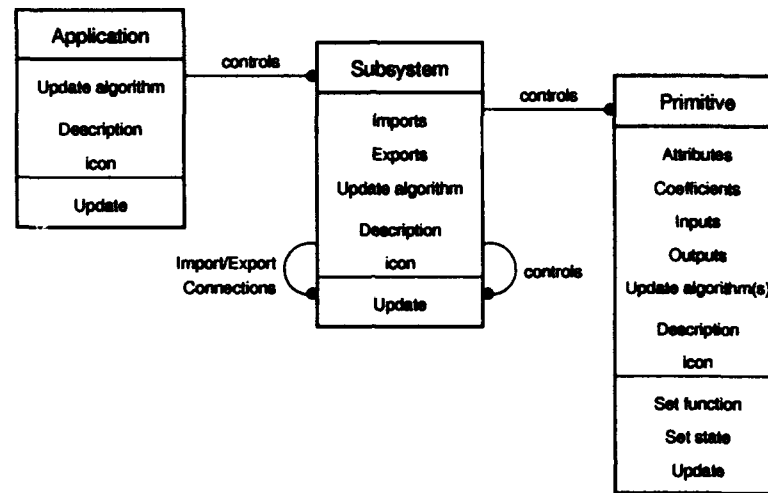


Figure 4.3 Architect Implementation of the OCU Model

This figure shows that an application controls one or more subsystems, subsystems control one or more primitives and/or subsystems (shown by the circular association “controls”), and a subsystem is connected to one or more (possibly including itself) subsystems through import/export connections. An application or subsystem can control several subsystems/primitives, but each subsystem/primitive can only be controlled by one application or subsystem. The export of one subsystem can be connected to several imports of other subsystems, but each import can only be connected to one export.

There are several differences between this model and the OCU model described in (20). Here, an application object is specified and has an update algorithm. The Stabilize, Initialize, and Configure procedures for a subsystem are not implemented at this time

⁴The object modeling notation used for this figure is defined in (16).

and are areas for future research⁵. The controller function of the subsystem is contained in the update algorithm. Objects are renamed to primitives. The create and destroy procedures for subsystems and primitives are not needed since the REFINED object base is persistent between executions (these objects do not need to be recreated in the object base every time the application is executed). The attributes, current_state, and constants for an object are all implemented as attributes in the primitives. Also, a description and icon were added to applications, subsystems, and primitives to allow domain visualization and a method for the Domain Engineer to pass information about the primitives to the Application Specialist. The differences that most affect this research are the lack of the Initialize procedure for subsystems and the implementation of object attributes, current_state, and constants as primitive attributes; these differences will be discussed in detail in the next chapter.

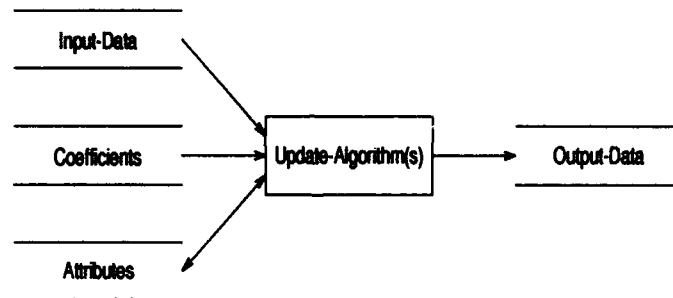


Figure 4.4 Primitive Data-Flow Diagram

Figure 4.4 shows a data-flow diagram for a primitive. Note that a primitive can change its own Attributes through the update algorithms. The Input-Data is provided through the connections to other primitives (i.e. it is copied out of other primitive's Output-Data). The SetState procedure can modify an object's Attributes directly, while the SetFunction procedure changes which Update-Algorithm the primitive uses to update its state⁶. The SetFunction procedure can also be used to change the values of the Coeffi-

⁵In Architect, the imports and exports are actually implemented as separate objects with associations tying them to their subsystems, but that is not important at this level of abstraction.

⁶In his research, Welgan added the capability for Architect to execute time-driven and event-driven applications. When using this capability, each primitive must have its own SetState procedure, and it is this procedure, not the Update-Algorithm, that modifies the Attributes and sets the Output-Data. It can also subsume some or all of the functionality of the Update-Algorithm.

cients. The Input-Data is supplied by the subsystem from its imports and the output data is incorporated into the subsystem's exports.

4.2.3 Creating an Application. Figure 4.5 shows the main control panel of AVSI. The buttons on the upper portion provide access to Architect's capabilities, while the Message Window reports success or failure (with error messages) of attempted operations along with other helpful information. Each of the buttons bring up other windows to accomplish its mission.

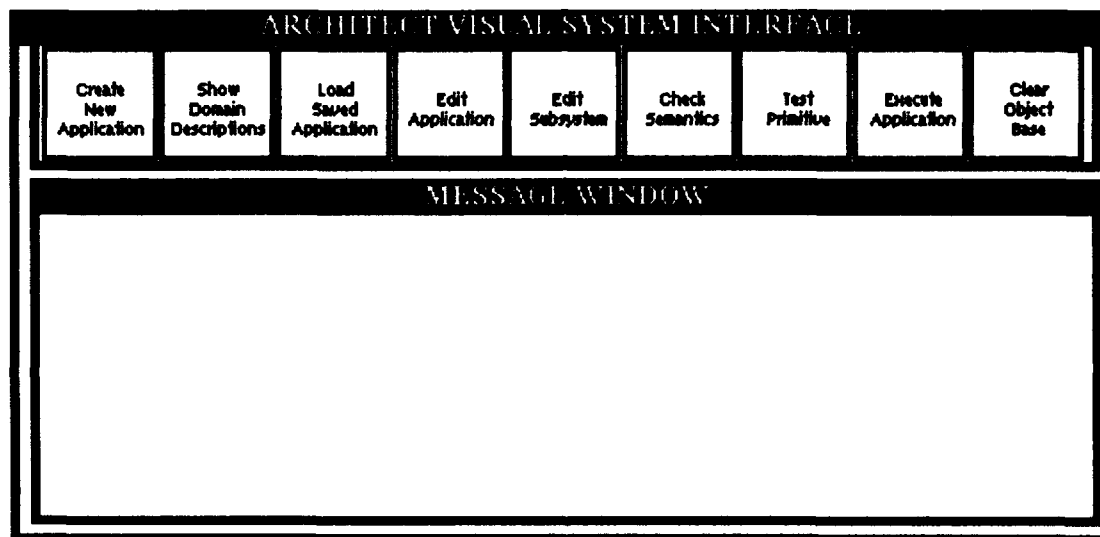


Figure 4.5 The Architect Control Panel

An application is built by adding subsystems to it and then building each of those subsystems. An example of the windows used to choose primitives to compose a subsystem is shown in Figure 4.6. The window on the right shows the available primitives for the chosen domain (in this case the digital circuits domain⁷); the window on the left shows the current subsystem (in this case a subsystem that provides an exclusive-or functionality using four NAND gates). Primitives are added to the subsystem by simply moving them (with a mouse) from the right window to the left. Subordinate subsystems can also be added in the left window (through a menu). It is important to note the ease of adding primitives and subsystems to applications—it only takes a few clicks on the mouse.

⁷Information on the digital circuits domain implementation in Architect is contained in (3).

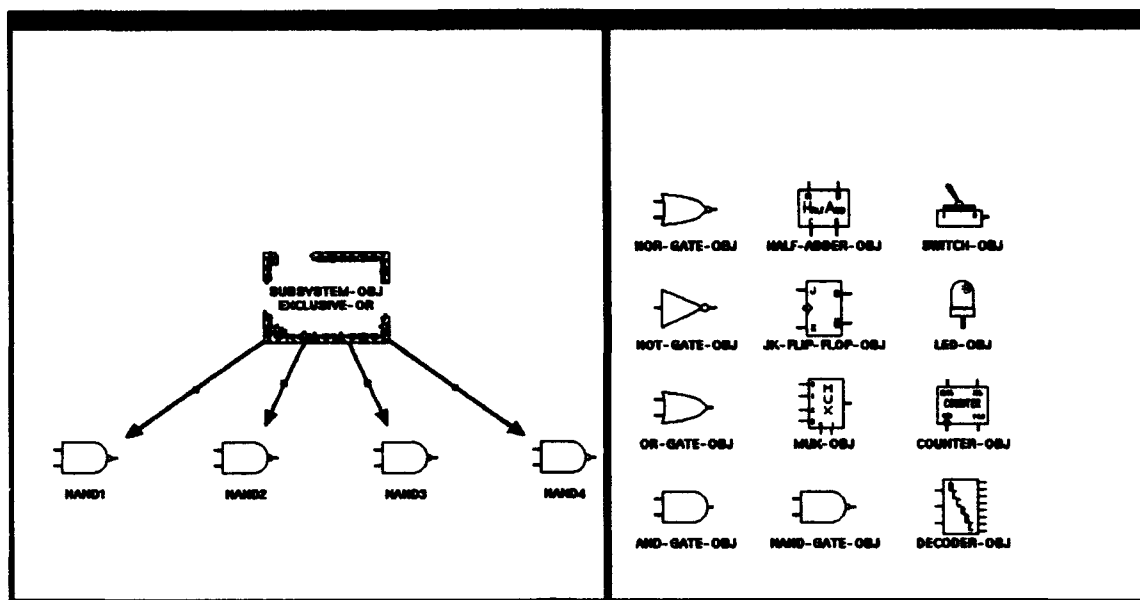


Figure 4.6 XOR Primitives and Circuits Domain Technology Base

After subsystems and primitives are added to an application, the connections between the subsystem/primitives need to be captured. The Application Specialist specifies these connections in two steps: connections between primitives in the same subsystem and connections between subsystems. An example of the method for the first step is shown in Figure 4.7. Note that proper positioning of the primitive icons in this screen can provide a visual cue to the function of this subsystem (in this case, an "exclusive or").

Finally, the update algorithms for the application and all subsystems need to be specified. This specification is also accomplished in a visual environment⁸. These algorithms control the order that the update functions of all subordinate subsystem/primitives are called (along with two other functions, SetState and SetFunction, that will be explained later). Conditional (if) and looping (while) statements can be included in these algorithms.

⁸See (9) for more information on how update algorithms are specified.

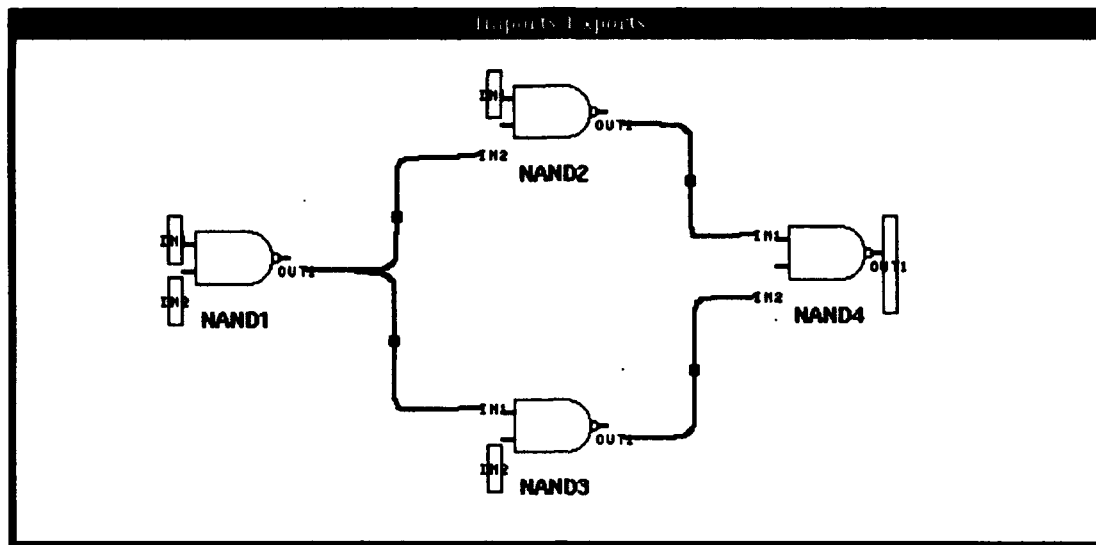


Figure 4.7 XOR Internal Connections

4.3 Technology Base Population Methodology for Architect

Now that Architect has been briefly explained, the instantiation of the generic Knowledge Base Population process described in Chapter III can be presented. This instantiation is shown in Figure 4.8.

4.3.1 Domain Analysis. The most obvious change in this figure from Figure 3.4 is that activities one and two are "grayed out" along with their properties (inputs, outputs, methods, and constraints). They are grayed out because, as stated in Chapter III, these two activities comprise the Domain Analysis process, which is independent of a particular DOACS (in this case Architect). Since these activities are independent, there should be no changes to them or their properties during the instantiation of the process for a particular DOACS.

Actually, the whole generic knowledge base population methodology is designed to be instantiated for a particular domain analysis method and DOACS. However, the independence between the domain analysis and domain implementation phases of this methodology allows us to instantiate activities three, four, and most of five for a given DOACS independent of any particular domain analysis method, while activities one, two, and part of

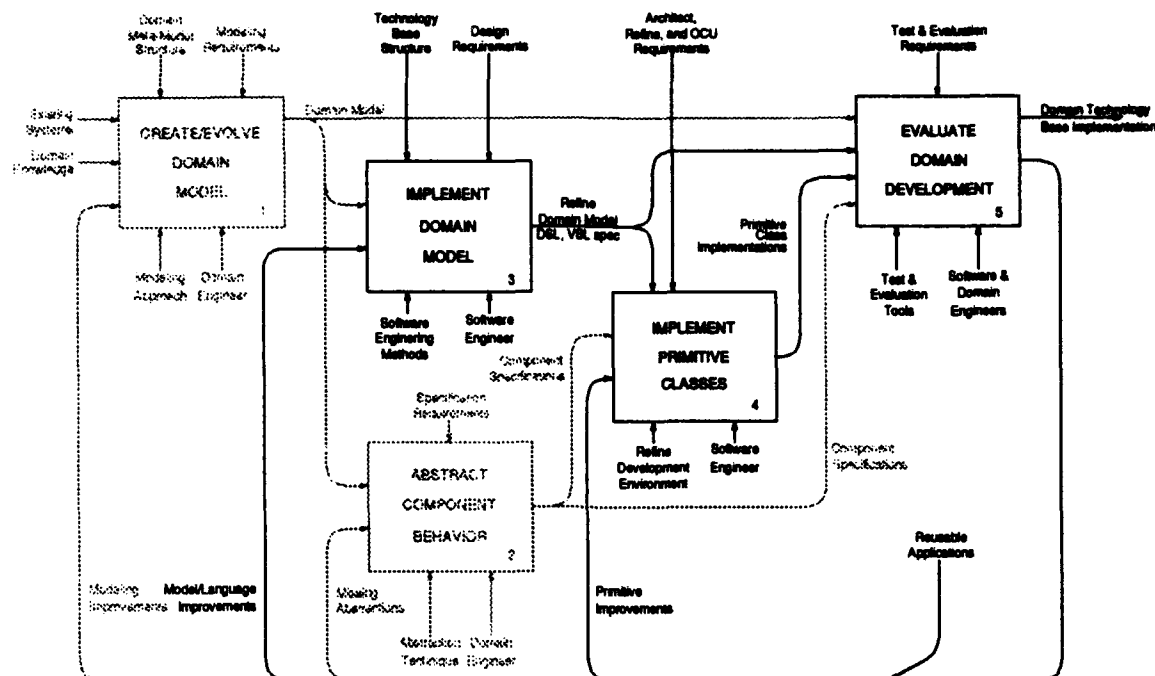


Figure 4.8 Technology Base Population Process

five can be instantiated for a given domain analysis method independent of any particular DOACS (actually, this is easier said than done, as explained in the next chapter). In this chapter, we only discuss the instantiation of this methodology for a particular DOACS, namely Architect.

Different domain analysis approaches can be used to implement different domains in the same DOACS, resulting in different instantiations of activities one and two. In fact, the domain analysis approach used should be chosen, at least in part, based on the characteristics of the domain to be analyzed. Another basis for which approach to choose is the availability of automated domain analysis tools that implement different approaches. The ideal situation would be to have an automated tool for domain analysis, and then create another tool that would transform the outputs of this domain analysis tool into the form required by the DOACS knowledge base under consideration. The domain analysis approach used for this research is described in Chapter V along with the implementation of a particular domain.

As the software engineering community continues to investigate domain analysis, standardized formats and automated tools to generate domain information in these formats should appear. One prototype domain analysis tool is OORA (Object-Oriented Requirements Analysis) Automated Knowledge System (OAKS) (10). OAKS was designed to capture domain information without regard to any particular DOACS. Using this tool, the Domain Expert defines object classes and the interrelationships between these classes; the advantage is that OAKS then checks the consistency and completeness of the entered information and displays any problems found.

4.3.2 Domain Implementation in Architect. Activities three and four comprise the domain implementation process and are instantiated for Architect in Figure 4.8. The next two sections describe these activities.

4.3.2.1 Implement Domain Model. Activity three in Figure 4.8 is the first step in implementing a domain in Architect. This activity implements the structure of the domain in the Technology Base. Currently, the Software Engineer accomplishes this activity by creating text files. There is ongoing research, described in (7), to populate the Technology Base from a database which, if implemented, will change the form of this activity. The type and amount of required information to implement a domain model will not change, but the method in which this information is input will change (i.e., filling in database forms instead of editing text files).

There are two constraints to the Implement Domain Model activity. The first is Architect's Technology Base structure. This constraint imposes the format for domain information that the Technology Base requires and requires that the implementation be accomplished using the REFINE language. The second constraint is the Design Requirements placed on domains implemented in Architect. Two examples of these Design Requirements for Architect are the OCU model and the file structure conventions listed in Appendix C. More examples of these two constraints are described later in this section, starting with the REFINE domain model.

A domain model in **REFINE** is defined in (34) as a class structure with attributes; the domain specific language (DSL) and visual specification language (VSL) (outputs of activity three in Figure 4.8 are called grammars and are not considered part of **REFINE** domain model. Our definition of a domain model in Chapter III, however, does include the DSL and VSL. In this section, as well as in Figure 4.8, we will use the words “**REFINE** domain model” to differentiate between these two uses of the term domain model.

REFINE Code Required to Implement a Domain in Architect. To implement a domain in Architect, the Software Engineer writes **REFINE** code to describe the domain in five parts. These five pieces of code:

1. declare the domain model classes
2. define the attributes for the domain classes
3. declare domain-specific types and functions
4. define the DSL
5. define the visual objects that are parsed in by the VSL (we refer to this code as simply the VSL)
6. declare the icon objects

The code that defines the **REFINE** domain model classes for the Digital Logic domain is shown in Figure 4.9. The primitive classes are capitalized to highlight them. Primitives in Architect are created from these classes. The three intermediate classes (Gates, Input-outputs, and Common-Circuits) are not strictly needed; all the primitive classes could be direct subclasses of Circuits. However, they do provide grouping information for the primitives when they are displayed in the domain Technology Base window (see the right half of Figure 4.6). Also, they aid in understanding the structure of, and logical groupings in, the domain. For example, it is easy to visualize an object model tree like Figure 4.10 from this code. In this figure, the link between classes denotes generalization (the upper class is a generalization of the lower classes) with inheritance (16). In Architect, primitives can only be instantiated from leaves in a tree like this one (this is an example of the Design Requirements constraint shown in Figure 4.8 for activity three).

```

var Circuits          : object-class subtype-of Primitive-Obj

var Gates              : object-class subtype-of Circuits
var  AND-GATE          : object-class subtype-of Gates %prim
var  OR-GATE           : object-class subtype-of Gates %prim
var  NOT-GATE          : object-class subtype-of Gates %prim
var  NAND-GATE         : object-class subtype-of Gates %prim
var  NOR-GATE          : object-class subtype-of Gates %prim
var Input-Output      : object-class subtype-of Circuits
var  SWITCH            : object-class subtype-of Input-Output %prim
var  LED               : object-class subtype-of Input-Output %prim
var Common-Circuits   : object-class subtype-of Circuits
var  COUNTER           : object-class subtype-of Common-Circuits %prim
var  DECODER           : object-class subtype-of Common-Circuits %prim
var  HALF-ADDER        : object-class subtype-of Common-Circuits %prim
var  JK-FLIP-FLOP      : object-class subtype-of Common-Circuits %prim
var  MUX               : object-class subtype-of Common-Circuits %prim

```

Figure 4.9 Architect's domain class code for the Digital Logic domain

The REFINe domain model is not complete, however, until the attributes of these classes are defined. Although inheritance is included in the REFINe object base, Architect requires that all attributes be pushed down to the primitive class level. Therefore, any attributes identified in the domain analysis for intermediate classes must be repeated in the lower classes. The code that implements attributes for the And-Gate class is shown in Figure 4.11. Each primitive class is required to have a REFINe attribute (implemented with the "map" structure) called Coefficients to implement the coefficient part of the OCU model (different from OCU attributes). The rest of this code implements the attributes identified during the domain analysis for the And-Gate (and all higher classes).

The domain-specific types define the type of information that will be passed between the primitives while the domain-specific functions are functions available for use by the primitives in their update functions. One main use of a domain-specific function is to add methods to manipulate the domain-specific types. There are no domain-specific types or functions for the Digital Logic domain, although more sophisticated domain will probably have some. Examples of these types and functions are presented in the next chapter during the discussion of the implementation of DSP domain.

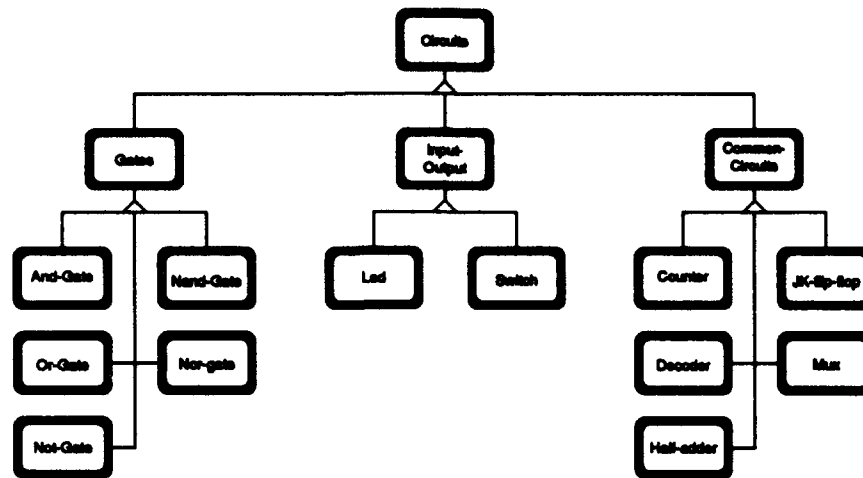


Figure 4.10 Digital Logic Class Structure

In this example, the And-Gate class has attributes of Delay, Manufacturer, Mil-Spec and Power-Level. Each attribute has a type and default value; this default value may be modified by the Application Specialist during composition. Each attribute can also have a description (the text in quotes above the attributes) that provides information to the Application Specialist when he/she is changing the value of that attribute. Each primitive class also has an attribute called "name", but it doesn't appear here since this is a default attribute for all REFINE objects.

After establishing the REFINE domain model, there are three more items that need to be defined for every domain. The first is the Domain Specific Language (DSL). Using this language (implemented through a grammar in REFINE), applications can be written to or parsed from text files. In fact, the Application Specialist could specify an application by creating a text file in the format specified by the DSL and then use the REFINE grammar tool called DIALECT to load the application. DIALECT would read the file and create the described objects in the Technology Base⁹. This was the method used by the Application Specialist to enter applications before AVSI was created; however, it is not as intuitive as AVSI and requires a more in-depth understanding of how Architect works. Figure 4.12 shows the portion, called a production, of the Digital Logic DSL for the And-Gate primitive

⁹ An example of one of these text files is contained in Appendix B.

```

var AND-GATE-COEFFICIENTS : map(AND-GATE, set(name-value-obj))
  computed-using
    AND-GATE-COEFFICIENTS(x) = {}

"The delay between the time the gate recieves its
input values and when it sets its output"
var AND-GATE-DELAY : map(AND-GATE, integer)
  computed-using
    AND-GATE-DELAY(x) = 0

"The name of the company that manufactured the gate"
var AND-GATE-MANUFACTURER : map(AND-GATE, string)
  computed-using
    AND-GATE-MANUFACTURER(x) = " "

"  true  -> the gate meets exacting military specifications
  false -> the gate only meets IEEE specifications"
var AND-GATE-MIL-SPEC? : map(AND-GATE, boolean)
  computed-using
    AND-GATE-MIL-SPEC?(x) = nil

"The ammount of power the gate requires"
var AND-GATE-POWER-LEVEL : map(AND-GATE, real)
  computed-using
    AND-GATE-POWER-LEVEL(x) = 0.0

```

Figure 4.11 Attribute code for the And-Gate primitive class


```

And-Gate-Obj      ::= ["and-gate" name
                        {["delay:" and-gate-obj-delay]
                        [{"is mil-spec" !! and-gate-obj-mil-spec? ] |
                        ["not mil-spec" ~!! and-gate-obj-mil-spec?]]]
                        ["manufacturer:" and-gate-obj-manufacturer]
                        ["power level:" and-gate-obj-power-level] } ]
                        builds And-Gate-Obj

```

Figure 4.12 And-Gate portion of the DSL

class. Comparing this production with the attributes for the And-Gate class shows that the DSL does not add any more information above that provided by the REFIN domain model—the attributes of the primitive class are just repeated in a different form. This is the case for all primitive classes implemented in Architect.

The second item that needs to be defined is a file that, when parsed in through the grammar that implements Architect's Visual Specification Language (VSL), creates the visual specification objects that AVSI requires to manipulate domain primitives. The acronym VSL is used (imprecisely) to refer to this code, although this code is really parsed through the already defined VSL grammar (a permanent part of Architect and not shown here). This VSL is used to load the information necessary to visualize the domain. An example of the portion of the Digital Logic file parsed by the VSL that implements the And-Gate primitive class visualization is shown in Figure 4.13. The "Icon" section shown in this figure is the same for every primitive, except for "and-bitmap-l" and "and-bitmap-s" which are the specific names of the icon objects for this primitive. The Edit section lists the attributes of the primitive class whose values can be changed by the Application Specialist (the attributes for the And-Gate primitive class were defined in Figure 4.11). If an attribute is not listed in this section, then this attribute will not be listed when the Application Specialist selects the menu choice that allows the values of attributes of a primitive to be changed. The decision whether or not to enter primitive class attributes in this section is based on information provided by the Domain Expert during Domain Analysis; in other words, the Domain Expert must specify which attributes the Application Specialist should be able to change.

```

attributes for AND-GATE are
Icon :
    label = class-and-name;
    clip-icon-label? = false;
    border-thickness = 0;
    bitmap4icon-1 = and-bitmap-1;
    bitmap4icon-s = and-bitmap-s
Edit :
    name : symbol;
    delay : integer;
    manufacturer : string;
    mil-spec? : boolean;
    power-level : real
end;

```

Figure 4.13 And-Gate portion of the VSL spec

The third item that needs to be defined for every domain is the icon object definitions. Architect creates the icons it uses from bitmaps, which are pixel-by-pixel representations of a graphic. The only information needed for these definitions is the icon object names (matching the ones entered in the Edit section described above) and the file names for the icon bitmaps. The portion of the code that creates these icon objects for the And-Gate is shown in Figure 4.14.

```

var and-bitmap-1    : any-type = (cw::read-bitmap( "CIRCUITS-TECH-BASE/andgate.icon-1"))
var and-bitmap-s    : any-type = (cw::read-bitmap( "CIRCUITS-TECH-BASE/andgate.icon-s"))

```

Figure 4.14 And-Gate primitive class icon object definitions

4.3.2.2 Create Primitive Classes. After the domain model is implemented as described above, the primitives need to be implemented. This implementation is activity four in Figure 4.8. In this activity, the behavior of the primitives will be implemented in an executable form. It is these behaviors that describe the functionality of the domain. As with activity three, the Software Engineer currently accomplishes this task by creating text files.

The two constraints on this activity (shown by the two arrows going into the top of the activity box in Figure 4.8) are the output of activity three and the requirements of Architect, REFINe, and the OCU model. The REFINe domain model defines the names

of the primitive classes and attributes; the same names must be used when implementing the primitive classes. An example of an OCU model requirement is that every primitive must have at least one update function. Several other Architect, REFINe, and OCU model requirements are described in the following paragraphs.

REFINe Code to Implement Primitives. The information needed to implement a primitive class consists of: inputs and outputs (name, type, and category), a description, at least one update function, and an icon for the class. The first three are implemented by the Software Engineer by writing REFINe code; the icons are created outside of the REFINe environment.

The inputs and outputs are the primitive's interface to the rest of the application. When a primitive is updated, it takes the current values of the inputs, along with the values of the coefficients and attributes, and uses the specified update function to calculate the outputs (see Figure 4.4); the update function(s) encapsulate the behavior of the primitive. The icon, which has two sizes (large and small), provides a visual cue for the function of the primitive class, as well as a method to use the mouse to add new primitives to an application. The icons for the Digital Logic domain were shown in Figure 4.6.

Figure 4.15 shows the portion of code that implements the And-Gate primitive class. This code is separated into three parts: inputs and outputs, primitive class description, and update function. The description can be viewed by the Application Specialist during the composition process. The description for the And-Gate primitive is intuitive, but in more complicated primitives the description can provide the Application Specialist with useful information, including design information that affects how the primitive behaves. There is only one update function for the And-Gate primitive class, called `Update1`, but if more than one existed it would be listed here. The last few lines in this figure set the default update function; this default can be changed by a `SetFunction` statement in the parent subsystem's update algorithm during execution.

The icons for primitive classes are currently made from bitmaps; the large from a 64x64 bitmap and the small from a 32x32 bitmap. The Software Engineer creates these bitmaps using any drawing tool that will output these sizes of bitmaps; therefore, this

```

%inputs/outputs-----
var AND-GATE-OBJ-INPUT-DATA : set(import-obj) =
  {set-attrs (make-object('import-obj),
    'import-name, 'in1,
    'import-category, 'signal,
    'import-type-data, 'boolean),

    set-attrs (make-object('import-obj),
    'import-name, 'in2,
    'import-category, 'signal,
    'import-type-data, 'boolean)}

var AND-GATE-OBJ-OUTPUT-DATA : set(export-obj) =
  {set-attrs (make-object('export-obj),
    'export-name, 'out1,
    'export-category, 'signal,
    'export-type-data, 'boolean)}

%set description-----

form set-and-gate-description
  re::zl-documentation(find-object('re::binding,'AND-GATE-OBJ)) <-
"The and gate takes two inputs and combines them together (after
the appropriate delay time) as follows:

  input1 | input2 | output
  -----|-----|-----
  T      | T      | T
  T      | F      | F
  F      | T      | F
  F      | F      | F
"

%update functions-----

function AND-GATE-OBJ-UPDATE1 (subsystem : subsystem-obj,
  and-gate : AND-GATE-OBJ) =

  let (in1 : boolean = get-import('in1, subsystem, and-gate),
    in2 : boolean = get-import('in2, subsystem, and-gate),
    the-output : boolean = false)

    the-output <- in1 & in2;
    set-export(subsystem, and-gate, 'out1, the-output)

%other update functions here, if any

var AND-GATE-OBJ-UPDATE-FUNCTION : map(AND-GATE-OBJ, symbol)
  computed-using
    AND-GATE-OBJ-UPDATE-FUNCTION(x) = 'AND-GATE-OBJ-UPDATE1

```

Figure 4.15 And-Gate primitive class code

information needs to be included in the domain analysis results. The bitmaps currently used in Architect were created by the OpenWindowsTM IconEdit tool shown in Figure 4.16. There are two sizes of icons because Architect windows can be rescaled to the smaller size when the icons start running off the edge of the screen. There are a few default icons that are supplied with Architect to aid the Software Engineer and standardize the visualizations of some of the common functions that appear in many domains¹⁰.

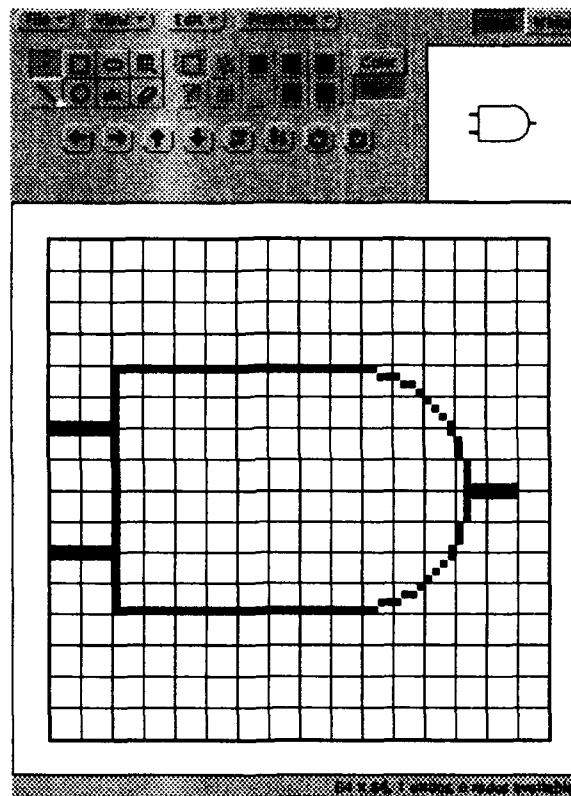


Figure 4.16 IconEdit tool showing the And-Gate icon

4.3.3 Evaluate Domain Development. As described in Chapter III, in the Evaluate Domain Development activity the Domain and Software Engineers evaluate the products they created during the other four activities. Therefore, the instantiation of this process will depend on the Domain Analysis method used as well as the Domain Imple-

¹⁰A list of the default Architect icons is contained in (9).

mentation process for the given DOACS. Other inputs into the instantiation of this activity is the standards of the organization and what automated tools are available.

This activity has one category of constraints, Test & Evaluation Requirements, which includes requirements from the domain analysis method and the DOACS, as well as organizational requirements and automated support requirements. An example of a domain analysis method requirement is a consistency check that ensures that all domain model object classes have a name and that these names are unique. An example of an organizational requirement would be to have all activity outputs reviewed by some quality control agent. Architect has several consistency checking requirements, but fortunately, most of them can be implemented automatically in REFINE. The most important non-automated consistency checking requirement for Architect is that the name of the primitive must be appended to the beginning of the attribute name when an attribute is implemented for a primitive (see Figure 4.11).

Test & Evaluation tools include both domain analysis tools and domain implementation tools. An example of an automated domain analysis tool is OAKS (described in Section 3.3). A special feature of OAKS is that it maintains a list of items the Domain Engineer has to enter/deconflict before the domain is consistent and complete. An example of a domain implementation tool is Architect's Test Primitive, described below.

Domain Implementation Evaluation. The evaluation of a domain implementation in Architect consists of three steps: evaluate the primitives, evaluate the domain model and primitive interfaces, and compare the domain implementation to the domain analysis results. The rest of this section describes these three steps.

As mentioned above, Architect has a tool to help evaluate individual primitives, which was implemented as a part of this research effort. This tool, called Test Primitive, is intended for use by the Software Engineer in testing individual primitives. It is provided on the Architect Control Panel for ease of use in this research environment, but would be removed if Architect was to be installed at a production site since use of this function should not be provided the the Application Specialist.

In Test Primitive, the Software Engineer can choose an individual primitive and test its update function(s). This is accomplished by providing windows in which the Software Engineer can change the values of the primitives attributes, inputs, and current update function directly, execute the primitive's specified update function, and then display the produced outputs. Of course, the use of this tool does not help if the Software Engineer does not have correct sample data to compare to the Test Primitive results; in other words, he/she must know what outputs to expect, or he/she will not know if they are correct. This sample data should be compiled by the Domain Engineer. An example of the use of the Test Primitive tool is discussed in section 5.5.2.

Although Architect does not have any specific tools to evaluate the domain model and primitive interfaces, the easy prototyping and execution capability of Architect provides a simple and convenient way to test them. After each individual primitive has been tested, the Software and Domain Engineers should compose some simple applications to test the interfaces between the primitives. It is impossible to compose all possible applications (since there are an infinite number of them), but enough should be implemented so that every primitive has been used and the Engineers feel confident that the domain semantics have been implemented correctly. During the composition and execution of these simple applications, the Software and Domain Engineers evaluate the domain structure as a whole, looking for things like missing primitives, conflicting primitives, primitives that overlap in functionality, etc.

Although there are many parts of the Evaluate Domain Development activity that can be clearly specified (such as specific consistency checks and automated tools described above), the Software and Domain Engineers should take a step back from the specific activities of this process and look at the domain analysis and domain implementation products as a whole. For the Software Engineer, we recommend that he/she take the time to sit down with the domain analysis and domain implementation results and compare them. He/she should not concentrate on the primitive behaviors, since they were tested already. The Software Engineer should, however, make sure the domain structure and interfaces are consistent and complete.

4.3.4 Reusable Applications. As described in Chapter III, there are two main methods to add applications into a knowledge base as new primitives. The first is to provide the application as it exists (with all its sub-components and connections) to the Application Specialist as a new reusable component and allow him/her to compose it into new applications just like the other components. The second is to feed this application back into the domain analysis process, abstract out its functionality, and then implement a new component with this behavior. The second method should be possible in any DOACS that fits into the G-DOACS framework (Figure 3.1); however, the first is more consistent with the reuse principle of DOACSs (and should be easier).

We can see the results of the second method in the Digital Logic domain in Architect. In Figure 4.9 there are three intermediate classes: Gates, Input-Output, and Common-Circuits. During the domain analysis process, the first primitives in the Gates and Input-Output classes were identified. The primitives in Common-Circuits were not identified until later when the domain had been in use for some time. In fact, the functions of the primitives were first built into applications by using the original set; then, as it became apparent that the functions of these applications would be used over and over, the functions of these reusable applications were implemented as individual primitives.

Architect cannot implement reusable applications using the first method (although there is a method that is somewhat similar using what are called generics (33)). A better way to implement the primitives in the Common-Circuits would have been to build them into a subsystem, test this subsystem, and then make it available to the Application Specialist as a new reusable component. This capability could be easily added to Architect; however, it would need to be integrated with the new data base implementation of the Technology Base being developed as a part of (7).

4.4 Summary

The purpose of this chapter was to provide a formal method for implementing a domain in Architect. This was accomplished by (after a brief description of Architect and its use of the OCU model) instantiating the general population method presented in Chapter III. This method includes implementing the domain model and then implementing

the individual primitive classes. These two activities are followed by a feedback loop in which the primitives and the domain as a whole are evaluated and compared to the domain analysis results, fixed, and verified and validated. Also, this feedback loop provides for the evolution of both the domain analysis results and the domain implementation.

V. Implementing the Digital Signal Processing (DSP) Domain in Architect

5.1 Introduction

In this chapter we discuss the analysis and implementation of the Digital Signal Processing (DSP) domain accomplished as a part of this research effort. We implemented the DSP domain in Architect to validate the Architect instantiation (presented in Chapter IV) of our generic knowledge base population methodology (presented in Chapter III). There are several reasons for this discussion: to show an example of domain analysis, to show an actual Architect domain implementation, and to provide a forum for explaining some details and problems of implementing a domain in Architect.

This chapter is organized along the lines of the population methodology in Figure 4.8; domain analysis, domain implementation, and domain evaluation of the DSP domain are discussed in that order, each in a separate section. The final section describes errors in the Architect system that affect the implementation and use of the DSP domain. First, however, we will provide a brief definition of the DSP domain.

5.2 The DSP domain

"Digital Signal processing deals with the representation of signals as ordered sequences of numbers and the processing of those sequences"(37:1). These signals are not always the typical electro-magnetic radiation, like radio signals, that first comes to mind; they can come from anything measurable, like temperature samples, the number of stars visible every midnight, or how long it takes to get dressed every morning. Each element of a signal is called a sample; a signal consists of an ordered sequence of samples. Signals can consist of real or imaginary numbers, and can be multidimensional (i.e., a sample can contain more than one number, but all samples in a signal must be the same size). Typical fields in which DSP has played a major role include speech and data communication, biomedical engineering, acoustics, sonar, radar, seismology, oil exploration, instrumentation, robotics, and consumer electronics, among many others (29:1). Typical reasons for processing these sequences of numbers include: estimation of characteristic signal param-

ters, elimination or reduction of unwanted interference, and transformation of a signal into a form that is in some *sense* more informative (37).

5.3 Domain Analysis

Since our work concentrates on domain implementation, the specific method we used to analyze the DSP domain is not particularly important to this research. However, it is briefly discussed here for four reasons. First, presentation of our DSP domain analysis process shows how the domain model and component specifications (the domain analysis outputs from Figure 3.4) are created so that they can be used in the domain implementation process. Second, the domain analysis methodology presented here is particularly useful in generating these outputs in a form that can be easily used in the domain implementation process for Architect. Third, even though identifying a particular domain analysis method is not important to this research, an example of a domain analysis process is helpful in understanding how to populate Architect's Technology Base. And finally, we present the domain analysis method we used to help show specific cases where the separation of the domain analysis process from a particular DOACS is not always best.

5.3.1 *The Independence Between Domain Analysis and a Domain Implementation.*

During the discussion of our generic knowledge base population methodology in Chapter III, we proposed that activities 1 and 2 in Figure 3.4 (which comprise the domain analysis process) should be independent of a particular DOACS (see Section 3.3.2). However, based on the results of our research, we conclude that this independence of domain analysis and domain implementation cannot always be accomplished and, even if it could, it might not be the most efficient method.

In Section 3.3 we compared the division between domain analysis and domain implementation to the division between the front and back ends of a compiler. This works well during compiler generation because, while there is not a single intermediate language, there are a few standardized ones; and these standard intermediate languages have the same basic functionality. The results of a domain analysis, however, are not fully understood, much less standardized. We believe that when domain analysis becomes more

fully understood and standardized, it can be separated from domain implementation as we propose (similar to the separation of the front and back ends of a compiler).

Even if the technology was currently available to implement this separation between the domain analysis and domain implementation phases, in some situations complete separation is difficult and not the best way to populate a knowledge base. For a small project (like this research effort), one person may take on the roles of both the Domain Expert and the Software Engineer. In this case, it is difficult, if not impossible, for that person to completely ignore his/her Software Engineer responsibilities during the domain analysis phase. The two phases should still be done separately, but the requirements of domain implementation will "bleed" over into domain analysis. Also, if the domain analysis is only going to be used to populate one knowledge base, then it can be accomplished faster and more efficiently if it is tailored to meet the requirements of the knowledge base. For example, if the DOACS does not have the capability for domain visualization, then it would be a waste of time to collect this information during domain analysis. Another example is that the information identified during domain analysis could be collected in a form based on the architecture(s) available in the DOACS. It is our opinion that this could represent a significant time savings during the population process; an example of this is identified in the following section. In any case, the change to our methodology to handle this is to add "domain implementation requirements" as an input to activities one and two in Figures 3.4 and 4.8.

5.3.2 The Domain Analysis Method Used. For this effort we used part of McCain's Software Development Methodology for Reusable Components as the basis of our domain analysis method. In (25), "a conventional product development model is used as a basis for a methodology for the construction of reusable software components" (25:70). Although his methodology does not deal with the population of knowledge bases, step two of his eight step process is a domain analysis method that fits well into our generic knowledge base population methodology.

There are five activities in this method:

1. Prepare Domain Information

2. Define reusable entities
3. Define reusable abstractions
4. Perform classification of reusable abstractions
5. For each component, perform a component domain analysis

Activities two through five are listed verbatim from the domain analysis part of McCain's method; activity one was added by this research. McCain's method has an additional step prior to the domain analysis step, called "perform market analysis", that covers some of what this added activity entails, but his method is oriented towards selling software, where we are more interested in basic domain analysis. Step one is from Prieto-Díaz's method discussed in Chapter III—it is the first step in Figure 3.2 (32:67). Note that the last activity in McCain's method maps directly onto the second activity of our generic population method (Figure 3.4).

McCain then goes on to list the activities that comprise a component domain analysis (25:74-76):

1. Define abstract interface specification. For each, identify:
 - Name
 - Description
 - Allowable values
 - Default value(s)
 - Error messages
2. Perform constraint analysis to minimize abstraction constraints
3. Define applicable algorithms and/or existing software
4. Define customization requirements based on the abstract interface specification
5. Define component visualization(s)

Again, these activities are listed verbatim from step two of McCain's method, except that we added the last one because McCain's process does not deal with component visualiza-

tions. These visualizations are needed to implement a domain in Architect. This addition illustrates how the DOACS's requirements can influence the domain analysis method.

As stated before, this method fits well into our generic knowledge base population method and its output is in a form that can be easily used to implement a domain in the current version of Architect. However, as capabilities are added to Architect, more and different types of information will have to be collected during the domain analysis process. For example, there are plans to add domain-specific semantic checks to Architect; this addition will require that domain-specific semantic rules be identified during the domain analysis process, which in-turn requires that a step be added to the above method to collect this information. If the state of the art was advanced enough to allow a standard domain analysis output to capture everything necessary in a domain, then an addition of more domain analysis steps like this one would never be needed.

5.3.3 Analyzing the DSP Domain. In this section we describe our analysis of the DSP domain. This discussion is organized according to the domain analysis method presented in the last section. In each step, some examples are given.

5.3.3.1 Prepare Domain Information. As stated in the previous section, we used Prieto-Díaz's Prepare Domain Information as the first activity in our domain analysis method. Prieto-Díaz breaks this activity down into five steps; the first four are applicable to our method ¹ (32:67):

- Define Domain Analysis Approach: our domain analysis method was presented in Section 5.3.2.
- Define Domain: a description of the DSP domain was presented in Section 5.2.
- Bound Domain: we identified several bounds for the DSP domain:

1. Discrete-time: "A discrete-time system is defined mathematically as a transformation or operator that maps an input sequence with values $x[n]$ into an output

¹Part of Prieto-Díaz's fifth step is to consolidate the outputs of the first four steps into a document, the rest is covered in later activities in our method.

sequence with values $y[n]$. This can be denoted as : $y[n] = T\{x[n]\} \dots$ " (29:17). In this representation, $x[n]$ and $y[n]$ denote input and output signals for the operation T . $X[i]$ (where i is a specific natural number) represents the value of a specific value (sample) of signal $x[n]$.

2. Deterministic: each sample of a signal is uniquely specified by a mathematical expression; in most cases, a real or complex number.
3. Finite-duration: signals are defined during some finite time interval and assumed to be zero outside that interval (some references, like (38), assume the values outside the specified time interval to be undefined, but this assumption is harder to implement in a computer).
4. One-dimensional: signals have only one dependent and one independent variable. If a signal was composed of temperature samples over time, then time is the independent variable and temperature is the dependent variable (from this point on we will assume time is the independent variable, remembering that other independent variables can be used). Multi-dimensional signal processing could easily be added to the domain by duplicating each component and adding a "2D" subscript to each of these duplicated component's names.
5. Normalized: the interval between any two sequential values of the independent variable is the same between signals. For example, if the independent variable is time, then two signals being added together are assumed to have the same elapsed time between each sample of all signals used in an application (for example, all signals are sampled at a rate of three samples per second).
6. Linear: for $T\{x_1[n]\} = y_1[n]$ and $T\{x_2[n]\} = y_2[n]$, T is linear if $T\{x_1[n] + x_2[n]\} = T\{x_1[n]\} + T\{x_2[n]\} = y_1[n] + y_2[n]$ and $T\{ax[n]\} = aT\{x[n]\} = ay[n]$. The first equality is called the additive property; the second is called the homogeneity or scaling property. All operations (and combinations of operations) for this domain model are linear.
7. Time-invariant (also called shift-invariant): a time shift or delay of the input signal causes a corresponding shift in the output signal.

8. Real: all signals are real, as opposed to signals that cannot exist in the real world such as the exponential signal $x[n] = e^n$ whose values increase forever.

These bounds are typical DSP restrictions. More detailed discussions can be found in (29) and (38), among others.

- Select Knowledge Sources:

1. Existing Systems: the three we found most useful were Khoros (40), PCDSP (2), and MatLab (21). The Joint-Modeling and Simulation System project (6) is developing DSP components, but we only had access to a portion of the Ada code. No documentation, descriptions, or domain analysis results were available. From this code, it appears that during the domain analysis process the DSP components were captured in a form that would facilitate implementation in Architect (in the code each DSP component is implemented in its own package, with inputs, outputs, attributes, and an update function just like in Architect). However, even if we had the complete set of J-MASS DSP Ada code, low-level language code is not a sufficient input for the Architect domain implementation process because, among other reasons, low level design decisions are made in creating code which should not influence domain analysis.
2. Experts: AFIT faculty and students conducting DSP research were consulted during the domain analysis and domain implementation activities.
3. Existing Documentation: several texts on DSP were used. The most helpful were (29), (38), (11), (19), (24), and (30).

5.3.3.2 Define Reusable Entities and Abstractions. In this activity, reusable entities and reusable abstractions are identified and defined. "Reusable entities are those entities for which a collection of related independently selectable reusable components may be defined...Reusable abstractions identify candidate independently selectable reusable components" (25:74). In simpler terms, the reusable abstractions are the components of the domain that will be used in composing applications, while the reusable entities are generalizations (higher level classes) of those components. Although it is not apparent

from McCain's definitions, he does allow for multiple levels of reusable entities (i.e. generalizations of generalizations).

We have listed the "define reusable entities" and "define reusable abstractions" activities together because not all object-oriented methodologies have them in the same order as McCain. In DSP domain analysis, we interleaved these two activities; in some parts of the domain we identified the entities first and then the abstractions for that entity, and in other parts we identified them in the reverse order. The results of this activity for the DSP domain are deferred until the description of the next activity.

5.3.3.3 Perform Classification of Reusable Abstractions. In the Perform Classification of Reusable Abstractions activity, the Domain Expert takes the components and generalizations (reusable entities and abstractions) from the previous activity and organizes them into an object model hierarchy. During this activity for the DSP domain, we found the existing systems listed in Section 5.3.3.1 to be particularly useful (they were also useful during the last activity). We used them by extracting their domain object model hierarchies and comparing them to each other and our hierarchy. The upper levels of these extracted hierarchies (down to, but not including, the components that would be the leaves of these trees) are shown in Figures 5.1 (40), 5.2 (2), and 5.3 (21). These figures use the object model notation introduced in Section 4.2.2.

The Khoros system (40) was the most helpful of the three, most likely because Khoros is a DOACS with similar capabilities to Architect. It should be noted that the DSP domain model hierarchy for MatLab includes only the parts listed in the Digital Signal Processing Toolbox (21); there are many other parts of MatLab (especially the arithmetic functions) that can be used in conjunction with this Toolbox.

The domain model hierarchy we developed for the DSP domain during this activity is shown in Figure 5.4. In the interests of space, the components are listed under each class instead of being shown in boxes linked to their parents. In general, the reusable entities identified in the previous activity become the interior nodes in the tree; the reusable abstractions become the leaves. In a domain as diverse and rapidly changing as DSP, no domain model can possibly include all aspects of the domain. In this domain model

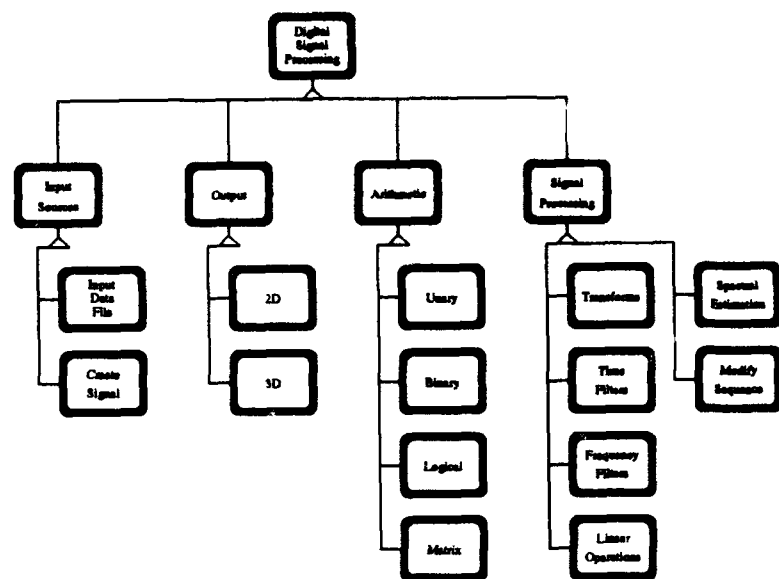


Figure 5.1 Khoros DSP domain model hierarchy

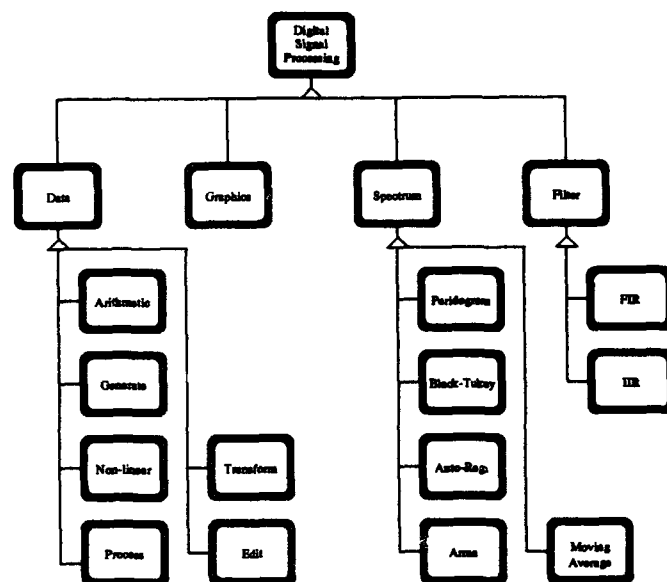


Figure 5.2 PCDSP domain model hierarchy

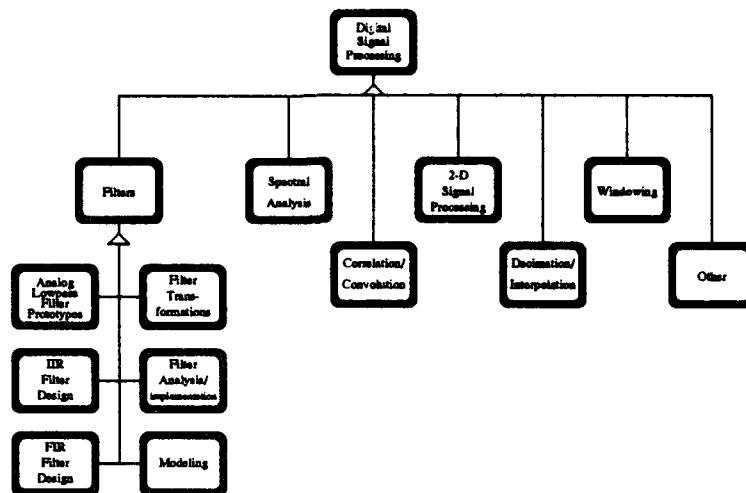


Figure 5.3 MatLab DSP domain model hierarchy

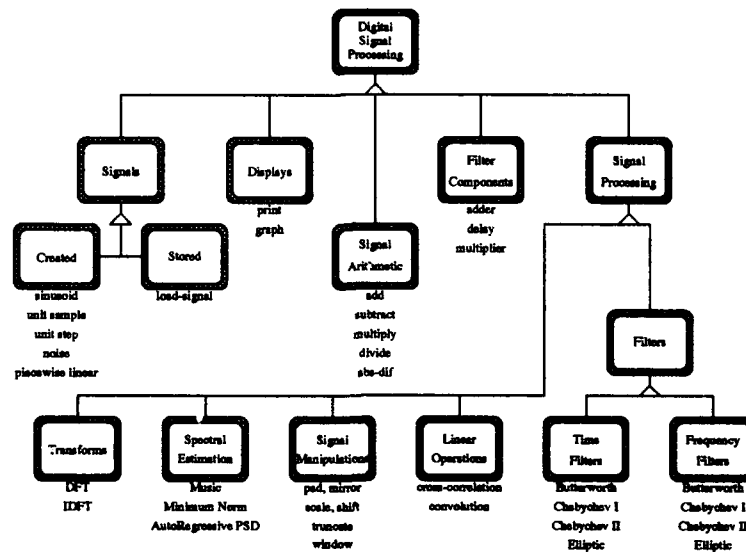


Figure 5.4 Architect DSP domain model hierarchy

hierarchy, only a canonical set of components have been identified; there are many others that could have been included.

We developed this domain model hierarchy iteratively (i.e. this is not the first hierarchy we developed) by comparing the information gathered from the knowledge sources listed in Section 5.3.3.1. After we developed our first full domain hierarchy, we held a mini-“design review” where we presented our results to a domain expert. Figure 5.4 shows the hierarchy after the domain expert’s suggestions were included.

Figure 5.4 shows the domain model hierarchy only; each item in this figure must also contain a definition and list of attributes. For example, the following definition and attributes were identified for the Discrete Fourier transform (DFT) component:

Name: Discrete Fourier transform (DFT).

Description: The Discrete Fourier transform (sometimes called the Discrete-Time Fourier transform in DSP) converts a signal from the time domain to the frequency domain. The independent variable of the resulting signal is radians and, no matter how many samples it consists of, the first sample is at 0 radians and the last is at 2π radians (the rest are evenly distributed between this range). For real signals, the output from 0 to π is the mirror of the output from π to 2π . The converse of the DFT is the inverse Discrete Fourier transform (IDFT).

Attributes: The DFT only contains one attribute:

- Name: Scale-by-N?²
- Type: boolean
- Description: when true, this attribute causes each sample of the output of the DFT to be scaled (divided) by N, where N is the number of samples in the input signal.

²The question mark is part of the name.

5.3.3.4 *Perform Component Domain Analysis.* Now that the domain model hierarchy has been created, each of the components must be specified in more detail. Again, we will use the DFT component as an example:

1. Define abstract interface: The DFT component has one input:

- Name: Input
- Description: a signal.
- Allowable values: any valid signal.
- Default value(s): none.
- Error messages: "input signal not defined" if the input signal has not been provided to the DFT.

The DFT component also has one output:

- Name: Output
 - Description: a signal.
 - Allowable values: any valid signal containing complex numbers.
 - Default value(s): none.
 - Error messages: none.
2. Perform constraint analysis: there are no constraints on our DFT component in general (other than those on the domain as a whole); however, different algorithms for computing a DFT have different constraints. For example, if we used the standard Fast Fourier transform (FFT) algorithm for calculating the DFT, the number of samples in the input signal would have to be a perfect square.


3. Define applicable algorithms: the DFT is specified by the equation

$X[k] = \sum_{n=0}^{N-1} x[n]W_N^{kn}$, where N is the number of samples in the input signal and $W_N = e^{-j(2\pi/N)}$ (in signal notation, $x[n]$ denotes a signal in the time domain while $X[n]$ denotes a signal in the frequency domain; the first sample of each is at $n=0$). The equalities $e^{\pm Aj} = \cos(A) \pm j * \sin(A)$ and $e^{\pm jx\pi} = e^{\pm j(x\pm 1)\pi}$ can also be useful. One pseudo-code algorithm to implement the DFT is:

```

let N be the size of the input signal
for i in 0 to N-1 do
  Output(i) = 0
  for j in 0 to N-1 do
    A = (2*pi*i*j)/N
    Output(i) = Output(i) + Input(j) * (cos(A) - j*sin(A))
  end(for)
  if Scale-by-N? then
    Output(i) = Output(i) / N
  end(if)
end(for)
return Output

```

4. Define customization requirements: the DFT component may be implemented as two components, one that takes a signal of real samples, and one that takes a signal of complex samples. The real sample case is the more common. Also, the attribute Scale-by-N? should be assigned at run-time to customize that particular DFT component.
5. Define component visualization(s): the DFT component does not have a commonly accepted visualization; however, there is a common visualization for a generic transform. Therefore, the DFT component may be visualized as: . There are no changes to this visualization to reflect state changes (since the DFT component has no state other than the value of its attribute).

Although the DFT component does not have a commonly accepted visualization, other components do. To find these visualizations, the Domain Expert should look for guidance from existing systems and documentation (books). Figure 5, which we found in (11:373), shows an example of the type of visualization information that the Domain Expert should look for.

5.4 Domain Implementation in Architect

In this section we describe our DSP domain implementation in Architect. This process consists of activities three and four in the Architect instantiation of our generic knowledge base population methodology (see Figure 4.8). These activities are discussed in the following two sections.

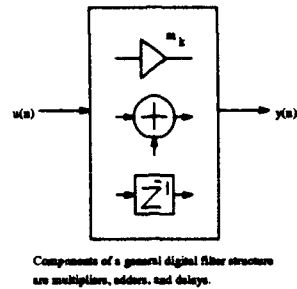


Figure 5.5 Common filter component visualizations

5.4.1 Implement Domain Model. Barring any problems found at this stage, if the domain analysis is detailed and consistent, then the Implement Domain Model activity of our process (described in Section 4.3.2.1) is trivial. First, the class hierarchy from Figure 5.4 is transformed into Architect domain class code. The code for the DSP domain is shown in Figure 5.6.

There are two reasons for the differences between the classes in Figure 5.4 and Figure 5.6. The first reason is that, due to time constraints, we decided not to implement all identified components. This also demonstrates the capability we discussed Section 3.3.1 to implement a portion of the domain, then use that portion to generate applications before the rest of the domain is implemented. The second reason is that Figure 5.4 was our first attempt at structuring the DSP domain; this structure evolved during our thesis effort. The final structure appears in Section 5.5, where we compare it to the original and discuss the differences.

The code that implements the domain-specific types and domain-specific functions is shown in Figure 5.7. In the next activity, Implement Primitive Classes, one of the items that must be specified for each primitive class is the type of the data being used in that input or output. If this information is other than one of the five basic REFINED types (boolean, integer, real, string, symbol), then a domain-specific type must be defined for this information. The main purpose of the domain-specific functions is to provide methods to manipulate the domain-specific types, although any function that will be used repeatedly in the primitive update functions should be defined here.

```

% domain model classes
var DSP : object-class subtype-of Primitive-Obj

var Signals : object-class subtype-of DSP
var Sinusoid : object-class subtype-of Signals %prim
var Stored-Signal : object-class subtype-of Signals %prim
var Unit-Sample-Sequence : object-class subtype-of Signals %prim
var Unit-Step-Sequence : object-class subtype-of Signals %prim
var Noise : object-class subtype-of Signals %prim
var Piecewise-Linear : object-class subtype-of Signals %prim
var Displays : object-class subtype-of DSP
var Print-Signal : object-class subtype-of Displays %prim
var Save-Signal : object-class subtype-of Displays %prim
var Graph-1-Signal : object-class subtype-of Displays %prim
var Graph-2-Signal : object-class subtype-of Displays %prim
var Graph-3-Signal : object-class subtype-of Displays %prim
var Graph-4-Signal : object-class subtype-of Displays %prim
var Signal-Arithmetic : object-class subtype-of DSP
var Signal-Adder : object-class subtype-of Signal-Arithmetic %prim
var Signal-Multiplier : object-class subtype-of Signal-Arithmetic %prim
var Signal-Subtractor : object-class subtype-of Signal-Arithmetic %prim
var Signal-Divider : object-class subtype-of Signal-Arithmetic %prim
var Signal-Abs-Dif : object-class subtype-of Signal-Arithmetic %prim
var Real-to-Complex : object-class subtype-of Signal-Arithmetic %prim
var Complex-to-Real : object-class subtype-of Signal-Arithmetic %prim
var Filter-Components : object-class subtype-of DSP
var Adder : object-class subtype-of Filter-Components %prim
var Delay : object-class subtype-of Filter-Components %prim
var Multiplier : object-class subtype-of Filter-Components %prim
var Input-Buffer : object-class subtype-of Filter-Components %prim
var Output-Buffer : object-class subtype-of Filter-Components %prim
var Signal-Processing : object-class subtype-of DSP
var Transforms : object-class subtype-of Signal-Processing
var DFT : object-class subtype-of Transforms %prim
var IDFT : object-class subtype-of Transforms %prim
var Spectral-Estimation : object-class subtype-of Signal-Processing
var Filters : object-class subtype-of Signal-Processing
var Time-Filter : object-class subtype-of Filters %prim
var Frequency-Filter : object-class subtype-of Filters %prim
% specific filters belong here
var Filter-Design : object-class subtype-of Signal-Processing
var User-Designed-Filter : object-class subtype-of Filter-Design %prim
%Butterworth, etc. filter designs belong here
var Linear-Operations : object-class subtype-of Signal-Processing
var Convolution : object-class subtype-of Linear-Operations %prim
var Signal-Manipulations : object-class subtype-of Signal-Processing
var Pad-Signal : object-class subtype-of Signal-Manipulations %prim
var Scale-Signal : object-class subtype-of Signal-Manipulations %prim
var Truncate-Signal : object-class subtype-of Signal-Manipulations %prim
var Window-Signal : object-class subtype-of Signal-Manipulations %prim
var Reverse-Signal : object-class subtype-of Signal-Manipulations %prim

```

Figure 5.6 Architect's domain class code for the DSP domain


```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
% domain-specific types

type sample-type          = real
type real-signal-type     = seq(sample-type)
type complex-sample-type  = tuple(real-part: sample-type,
                                imaginary-part: sample-type)
type complex-signal-type  = seq(complex-sample-type)
type filter-design-type   = tuple(a: seq(real), b: seq(real))

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%domain-specific functions
function complex-add(X1:complex-sample-type,
                    X2:complex-sample-type) : complex-sample-type =
  <X1.real-part + X2.real-part, X1.imaginary-part + X2.imaginary-part>

function complex-multiply(X1:complex-sample-type,
                          X2:complex-sample-type) : complex-sample-type =
  <X1.real-part * X2.real-part - X1.imaginary-part * X2.imaginary-part,
    X1.real-part * X2.imaginary-part + X1.imaginary-part * X2.real-part>

function complex-divide(X1:complex-sample-type,
                        X2:complex-sample-type) : complex-sample-type =
  let(sum-squ : real = X2.real-part * X2.real-part + X2.imaginary-part * X2.imaginary-part)
  <(X1.real-part * X2.real-part + X1.imaginary-part * X2.imaginary-part) / sum-squ,
    (X1.imaginary-part * X2.real-part - X1.real-part * X2.imaginary-part) / sum-squ>

function complex-of(M:real, P:real) : complex-sample-type =
  <coerce(M * cos(P), 'single-float),
    coerce(M * sin(P), 'single-float)>

function Magnitude-of(C:complex-sample-type) : real =
  sqrt(C.real-part * C.real-part + C.imaginary-part * C.imaginary-part)

function Phase-of(C:complex-sample-type) : real =
  let(temp:real = 0.0)
  C.real-part ~= 0.0 --> temp = coerce(atan(C.imaginary-part/C.real-part),'single-float);
  temp

```

Figure 5.7 User-defined types and functions for the DSP domain

Next, the code to implement the attributes for each component, specify the DSL, and describe the visual objects (VSL) is created³. The code to implement these portions of the domain model for the DFT component appears in Figures 5.8, 5.9, and 5.10. Note that after the attributes (the DFT primitive class there has one attribute) are specified in the attribute section of the code, they are just repeated in the other two portions. The only case in which this repetition does not hold is if a primitive class has attributes that the user should not be able to change (for example, attributes that hold state data); these attributes are prevented from being edited by excluding them from the VSL. Remember that the text above each attribute is displayed in the window in which the Application Specialist can change the attribute values.

```
var DFT-COEFFICIENTS : map(DFT, set(name-value-obj))
  computed-using
    DFT-COEFFICIENTS(x) = {}

    "If this attribute is set to true, then each sample of the output
    will be divided by the number of samples in the input."
var DFT-SCALE-BY-INVERSE-N : map (DFT, boolean)
  computed-using
    DFT-SCALE-BY-INVERSE-N(x) = false
```

Figure 5.8 Attribute code for the DFT primitive class

```
dft ::= ["dft" name
        {(["scale" !! dft-scale-by-inverse-n] |
          ["no scale" ~!! dft-scale-by-inverse-n])}]
  builds dft,
```

Figure 5.9 DFT portion of the DSL

We have shown this code here because we have been using the DFT primitive class as an example throughout the description of the population process. However, since the DFT only has one attribute, it is not very interesting. Therefore, we also show this code for the Sinusoid primitive class, which has several attributes, in Figures 5.11, 5.12, and 5.13

In these figures it is much easier to see how attributes are repeated in each portion. This repetition lends itself to automation. It would be relatively easy to create a tool in

³From Chapter IV, DSL = Domain Specific Language and VSL = Visualization Specification Language.

```

attributes for dft are
Icon :
    label = class-and-name;
    clip-icon-label? = false;
    border-thickness = 0;
    bitmap4icon-l = dft-l;
    bitmap4icon-s = dft-s
Edit :
    name : symbol;
    scale-by-inverse-n : boolean
end;

```

Figure 5.10 DFT portion of the VSL spec

```

var SINUSOID-COEFFICIENTS : map(SINUSOID, set(name-value-obj))
    computed-using
        SINUSOID-COEFFICIENTS(x) = {}

"the number of samples the output will contain"
var SINUSOID-NUMBER-OF-SAMPLES : map(SINUSOID, integer)
    computed-using
        SINUSOID-NUMBER-OF-SAMPLES(x) = 64

"this value is half the difference between the maximum
and minimum samples in the signal"
var SINUSOID-AMPLITUDE : map(SINUSOID, real)
    computed-using
        SINUSOID-AMPLITUDE(x) = 1.0

"IMPORTANT: if a frequency > 0.5 is entered, aliasing
will occur and a signal with frequency < 0.5 will be generated."
var SINUSOID-FREQUENCY : map(SINUSOID, real)
    computed-using
        SINUSOID-FREQUENCY(x) = 0.125

"the phase shift, in radians"
var SINUSOID-PHASE-SHIFT : map(SINUSOID, real)
    computed-using
        SINUSOID-PHASE-SHIFT(x) = 0.0

"this offset will be added to every sample in the signal"
var SINUSOID-MAGNITUDE-OFFSET : map(SINUSOID, real)
    computed-using
        SINUSOID-MAGNITUDE-OFFSET(x) = 0.0

```

Figure 5.11 Attribute code for the Sinusoid primitive class

```

sinusoid ::= ["sinusoid" name
{["# samples:" sinusoid-number-of-samples]
["amplitude:" sinusoid-amplitude]
["frequency:" sinusoid-frequency]
["phase shift:" sinusoid-phase-shift]
["magnitude-offset:" sinusoid-magnitude-offset]]}
          builds sinusoid,

```

Figure 5.12 Sinusoid portion of the DSL

```

attributes for sinusoid are
Icon :
    label = class-and-name;
    clip-icon-label? = false;
    border-thickness = 0;
    bitmap4icon-l = sinusoid-l;
    bitmap4icon-s = sinusoid-s
Edit :
    name : symbol;
    number-of-samples : integer;
    amplitude : real;
    frequency : real;
    phase-shift : real;
    magnitude-offset : real
end;

```

Figure 5.13 Sinusoid portion of the VSL spec

which the Software Engineer would enter the attributes and whether or not each one is allowed to be changed by the Application Specialist. The tool would then generate this code. As a matter of fact, this tool should be built to cover the whole Architect domain implementation process (see Section 6.3).

The final step in implementing the domain model is to create the code that tells the AVSI where the icon files are for the different primitive classes. The portion of this code for the DFT primitive class is shown in Figure 5.14. The actual icons are created in the during the next activity (Implement Primitive Classes), but because of the Architect file conventions (see Appendix C), it is easier to specify this code for the whole domain at the same time, rather than adding to this code individually for each primitive class.

```
var dft-l: any-type = (cw::read-bitmap("DSP-TECH-BASE/dft.icon-l"))  
var dft-s: any-type = (cw::read-bitmap("DSP-TECH-BASE/dft.icon-s"))
```

Figure 5.14 DFT primitive class icon object definitions

5.4.2 Implement Primitive Classes. After the domain model is implemented, the the code to implement the interface (inputs and outputs), class description, and update function(s) for each primitive class is created. The code for the DFT primitive class appears in Figure 5.15.

Due to the form of the Domain Analysis outputs, the creation of this code simply involved converting pseudo-code into REFINE code. We captured the component functionality during the Domain Analysis process in this fashion (with only one function, which is based on transforming inputs to outputs) because we knew the form that the domain implementation would take. This is another example of the DOACS knowledge base structure affecting the Domain Analysis process.

The input/output type tell Architect what type of data that input/output contains. This type must be one of the basic REFINE types (boolean, integer, real, character, string, and symbol) or one of the user defined types entered implemented during the Domain Implementation activity.

```

%inputs/outputs-----
var DFT-INPUT-DATA : set(import-obj) =
    {set-attrs (make-object('import-obj),
                        'import-name, 'input,
                        'import-category, 'real-signal,
                        'import-type-data, 'real-signal-type)}

var DFT-OUTPUT-DATA : set(export-obj) =
    {set-attrs (make-object('export-obj),
                        'export-name, 'output,
                        'export-category, 'complex-signal,
                        'export-type-data, 'complex-signal-type)}

%set description-----

form set-and-gate-description
    re::zl-documentation(find-object('re::binding,'DFT)) <-
"The Discrete Fourier transform takes a signal in the time domain and
converts it to the frequency domain."

%update functions-----
function DFT-UPDATE (subsystem : subsystem-obj,
                    the-dft : DFT) =

    format(dsp-debug, "DFT-UPDATE on ~s~%", name(the-dft));

    let (S : real-signal-type = get-import('input, subsystem, the-dft),
        0 : complex-signal-type = [],
        angle : real = 0.0,
        the-size : integer = 0,
        cn : complex-number = <0.0, 0.0>)
    the-size <- size(S) - 1;
    (enumerate i from 0 to the-size do
        cn.real-part <- 0.0;
        cn.imaginary-part <- 0.0;
        (enumerate j from 0 to the-size do
            angle <- (2 * pi * i * j) / the-size;
            cn.real-part <- coerce(cn.real-part + S(j+1) * cos(angle),'single-float);
            cn.imaginary-part <- -1 * coerce(cn.imaginary-part +
                S(j+1) * sin(angle),'single-float));
        if DFT-SCALE-BY-INVERSE-N(the-dft) then
            cn.real-part <- cn.real-part / the-size;
            cn.imaginary-part <- cn.imaginary-part / the-size;
            0 <- append(0, cn)
        else
            0 <- append(0, cn));
    set-export(subsystem, the-dft, 'output, 0)

var DFT-UPDATE-FUNCTION : map(DFT, symbol)
    computed-using
    DFT-UPDATE-FUNCTION(x) = 'DFT-UPDATE

```

Figure 5.15 DFT primitive class code

There is only one piece of information that the Software Engineer must create during this whole domain implementation process (all the rest of the needed information is provided in the domain analysis results): the category names for the inputs and outputs. These category names are used by Architect during the semantics check to ensure that two connected components are allowed to be connected. If the category names are not the same for a connected input and output, then an error message is displayed and the application cannot be executed. There are no restrictions on these names, except that they must contain no spaces (they are not case-sensitive). Therefore, the Software Engineer can name them anything he/she wants (hopefully something descriptive of the data they contain), but the names must be consistent across all primitive classes. In the DSP domain there are four categories of inputs/outputs: real-signal, complex-signal, sample, and filter-design.

After the primitive implementation code is generated, the last step is to create the primitive class icons. Currently, these icons are being generated using the IconEdit tool shown in Figure 4.16. As stated in Section 4.3.2.2, two separate icons need to be created for each primitive class: one large (64x64) and one small (32x32). The large icons for the DSP domain are shown in Figure 5.16, which is a screen capture of the DSP Technology Base Window.

5.5 Evaluate Domain Development

The above discussions of the first four activities accomplished during the Architect DSP implementation process were written as if they had been done all at once without any mistakes; however, this is not the case. Our generic knowledge base population methodology process was designed to be an iterative process (discussed through-out Chapter III); it is impractical to expect that these four activities could be consistent, complete, and correct the first time through. Also, both the domain analysis results and the domain implementation should be evolved over time; therefore, the outputs of the first four activities are evaluated in this final activity. These evaluations feed back into the other four activities to provide this evolution.

In this section, we will discuss the evaluation and evolution of the domain model hierarchy and the testing of the individual primitives. The testing of the domain imple-

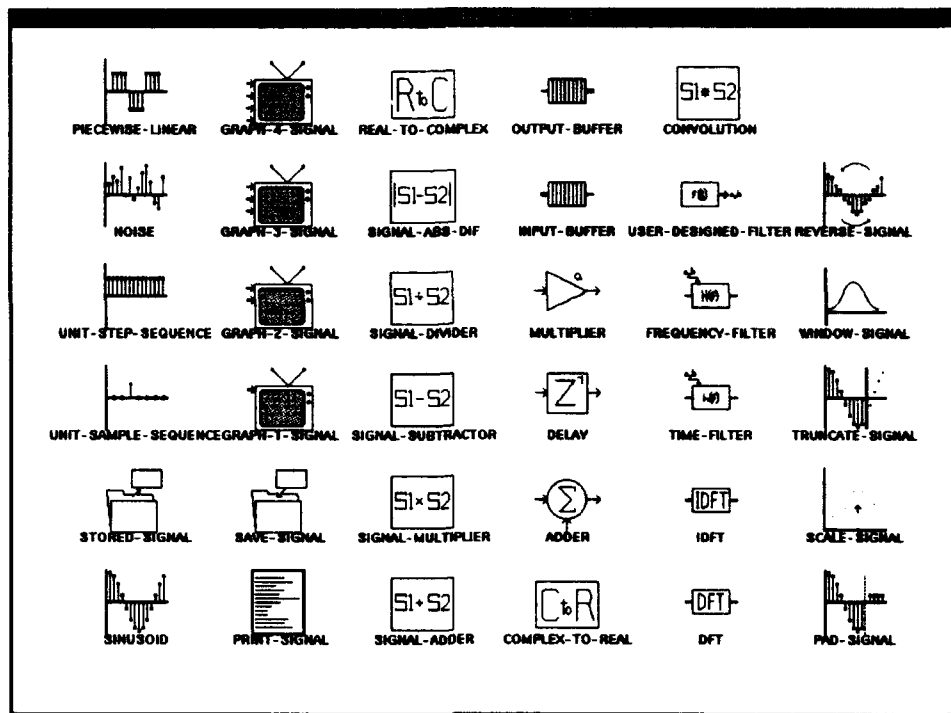


Figure 5.16 The DSP Technology Base Window

mentation, as specified in Section 4.3.3, was accomplished by creating “simple” applications that contain only a few primitives whose correct outputs were known. Some of these applications are presented in Appendix B.

5.5.1 Evolving the DSP Domain Hierarchy. As stated previously, the DSP domain model hierarchy shown in Figure 5.4 was the first attempt at classifying the domain. Figure 5.17 shows the final DSP domain model hierarchy developed as a part of this research (the main differences will be explained in the following paragraphs). It is expected that this hierarchy, along with the rest of the domain information, will continue to develop over time.

The changes made to the DSP domain model hierarchy were made for three main reasons. The first was to make the domain model more complete; more Transforms and Signal Manipulations components were added for this reason. However, with a complex and evolving domain like DSP, it will not be possible to add all possible components to

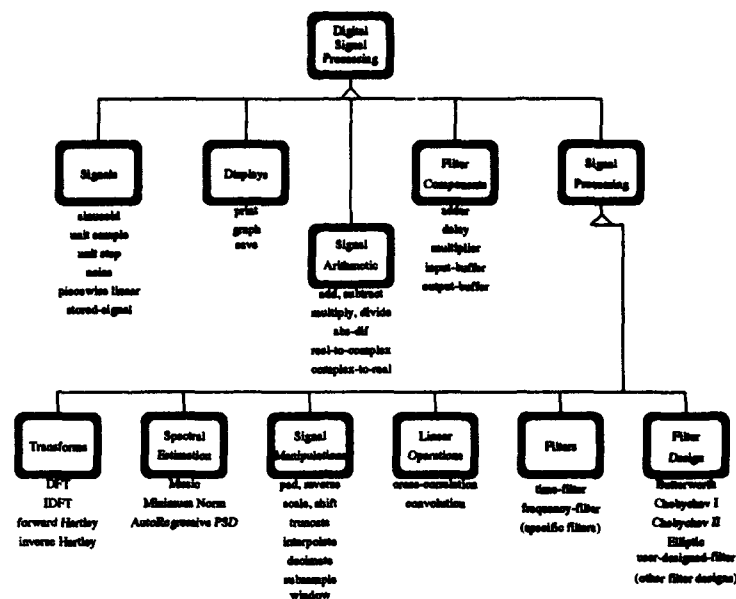


Figure 5.17 Revised Architect DSP domain hierarchy

the model. The Domain Engineer should resist the temptation to add components just because they can be added: only those components that are expected to be of use should be added. If a missing useful component is identified later, it can easily be added.

The second reason is that, as our understanding of the DSP domain grew, we decided to restructure the DSP model classes. The best example of a change for this reason is the restructuring of the filter portion of the model hierarchy (the lower right-hand corner of Figures 5.4 and 5.17). After more research into how filters were created, we decided that, while the structure in Figure 5.4 is a valid way of ordering filter components, the structure in Figure 5.17 is better. In this structure, an Application Specialist can create a filter by first choosing a generic time or frequency filter (which have attributes for specifying lowpass, highpass, bandpass, or bandstop) and then incorporating one of the standard filter designs, or by choosing a specific pre-designed filter (represented by "specific filters" in Figure 5.17). We do not attempt to list any pre-designed filters here because there are an infinite number of them. They should be implemented as the need for them is identified (we did implement one, the window-lowpass-filter, to validate the concept). In both model

hierarchies, the Application Specialist can also create a filter by using the primitives listed under "filter components".

The third reason for the changes between our starting and current DSP domain model hierarchies is that missing components were identified during the domain implementation and even during the use of the domain in Architect as application were composed. We identified that the input/output buffers under the Filter Components class were needed during the domain implementation phase. Most of the DSP components in our model have inputs and outputs consisting of signals, but the components under Filter Components (adder, delay, and multiplier) have inputs/outputs that consist of a single sample. Therefore, if these components are going to be used with other components (for example, Signals and Displays components), then there must be some intermediate components that convert between these two types. For this reason, the input/output buffers were added. We did not discover that the real-to-complex and complex-to-real components (under the Signal Arithmetic class) were needed until we were actually composing test applications. Specifically, we wanted to see a graph of the output of a DFT primitive, but this was not possible because the DFT primitive class outputs a complex-signal while all the Display primitive classes need a real-signal for their inputs.

The above paragraphs describe the evolution of the DSP domain model hierarchy during our thesis effort. The components defined in the Domain Analysis process, the DSP domain model implementation, and the DSP Architect primitives also evolved during this time. The evolution of the defined Domain Analysis components and DSP domain model implementation, while significant, are not as interesting or readily identifiable, and so will not be discussed here.

5.5.2 Testing Primitives. Architect has a tool for testing the behavior of primitives, called Test-Primitive. Figure 5.18 shows this tool being used to test the Complex-To-Real primitive.

The Test-Primitive function controls five windows. The first, shown in the upper right, is titled Test-Primitive and displays the name and icon of the primitive under test, along with buttons to update the primitive and exit this tool. The window below this

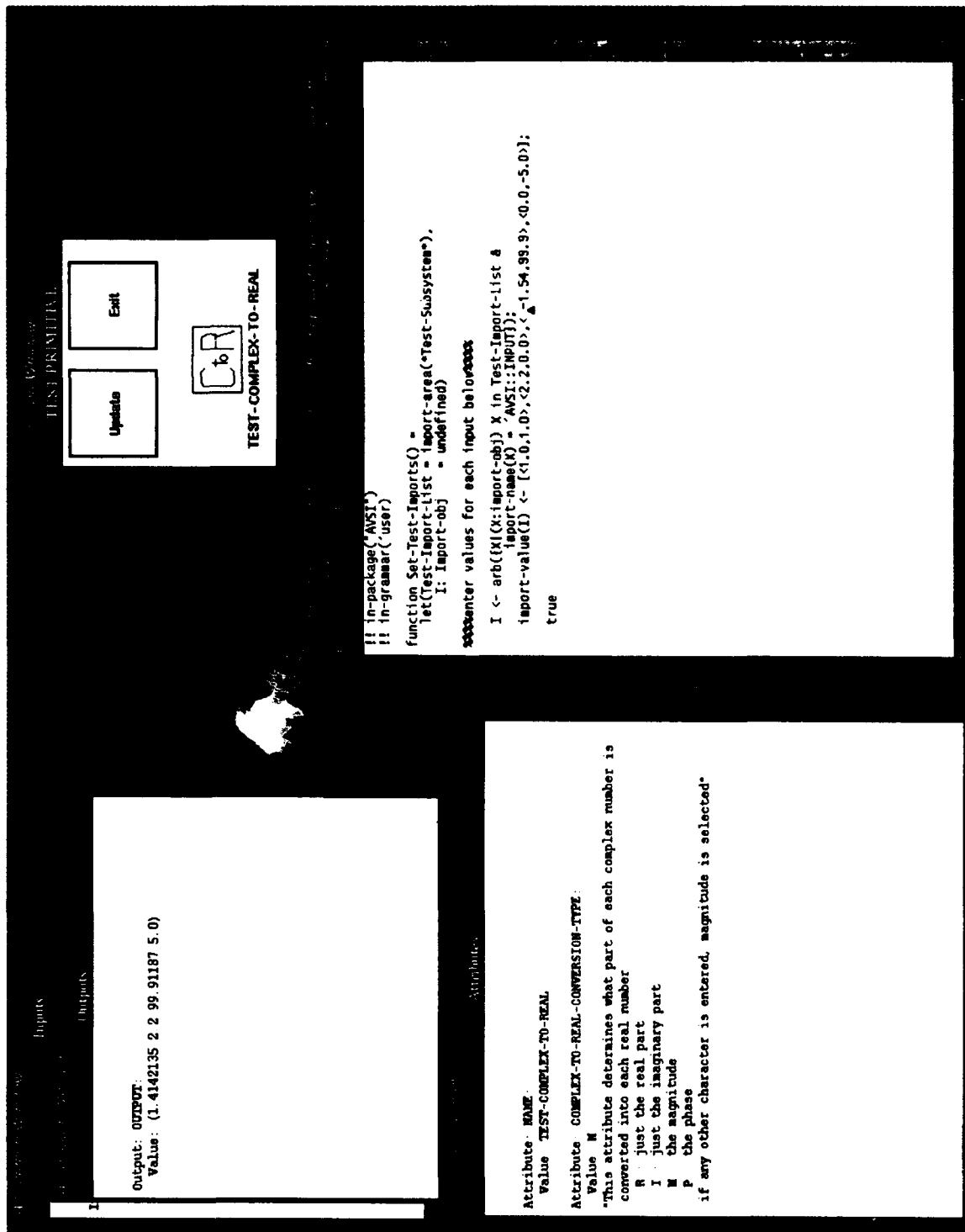


Figure 5.18 The Test-Primitive Function

one is a text-edit window containing automatically generated REFINE code in which the Software Engineer enters the desired test-case inputs (a separate section is created for each input – in this case there is only one). The window in the lower left, titled Attributes, lists the current values of the attributes for the primitive (in this case there is only one) and allows them to be changed. The Outputs window, immediately above the Attributes window, shows the outputs of the primitive (again, in this case there is only one) after it has been updated. The window behind this one, titled Inputs, is not necessarily needed—it just shows the current values of the inputs as a verification that the inputs entered in the text-edit window were indeed correctly interpreted and assigned to the appropriate inputs.

The Test-Primitive function is used by entering the desired test-case attribute and input values, choosing “Update” from the Test-Primitive window, and then comparing the outputs to the expected test-case outputs. Assuming these steps are accomplished correctly, a difference between the listed outputs and the expected test-case outputs demonstrates an error in the primitive. The absence of an error cannot be demonstrated; therefore, the software engineer should use good testing techniques in choosing the test cases to be examined (boundary testing, decision-point testing, etc.).

Once all the primitives have been tested separately (as described above for the DFT primitive), the interface between the primitives needs to be tested. This was done by creating some small applications whose expected behavior was already identified. Examples of two of these applications can be found in Appendix B. Some of these applications were validated by calculating the expected outputs for specific inputs by hand, while others were validated by building a similar application in Khoros and MatLab and comparing the results.

5.6 Architect Problems that Affect the Use and Implementation of the DSP Domain

Architect is an evolving system; the AFIT KBSE group continues to identify deficiencies in and new composition methods for Architect. This section discusses four of the biggest problems with Architect that have an impact on the implementation and use of the DSP domain in Architect.

5.6.1 "State" Attributes. The OCU model (described in Section 4.2.1) lists several parts of a component, but Architect does not implement them all separately. Specifically, OCU component attributes, *state_data*, and constants are all implemented in Architect as primitive attributes, with no method to distinguish between them. This combination of attributes and *state_data*, in particular, can cause problems in DSP applications (or any other domain, for that matter). The problem is that OCU component *state_data* should be set to some specific value before each execution (initialized), while OCU component attributes are set by the user and not initialized. Architect primitive attributes, however, cannot be initialized. This can result in effectively non-deterministic behavior, since the starting state of the primitives in Architect cannot be guaranteed to be the same between each execution.

An example of a DSP primitive that demonstrates this problem is the delay primitive (under the Filter Components in Figure 5.4). During each update, this primitive takes in a signal sample and returns the sample taken in during the previous update. To hold this value in between updates, the delay primitive has an attribute called *last-sample* (it is not listed in the VSL specification file, and therefore cannot be modified by the Application Specialist). Since there is no previous update to the first update, the delay primitive should return a zero for that first update. To implement this, the starting value of *last-sample* is zero. This works for the first execution of an application; however, on subsequent executions of that same application, the starting value of *last-sample* is whatever it happened to be set to during the last update of the primitive in the previous execution: there is no automatic method to reset it to zero! To work around this problem, the Application Specialist must remember to reset this attribute to zero for every delay primitive in the application before every execution, or the output for that execution will be invalid. Another work-around is to add a *SetState* call for each delay primitive in its parent subsystem's update algorithm, but this cannot currently be done through the visual system (AVSI). The problem is discussed in the next section.

5.6.2 Subsystem Update Algorithm. The AVSI does not currently provide a method for entering all the allowable statements into the update algorithms of subsystems

and applications. Specifically, `SetState`, `SetFunction` statements cannot be entered; also, there is no method to enter primitive output values into the expressions of "if" and "while" statements. The current work-around is to create the application, save the application, edit the application save file to add these statements, and then reload the application.

An example of this limitation affecting our research surfaced in an application containing Filter Components primitives. In general, such an application has a subsystem that contains an input-buffer, some number of adders, delays, and multipliers, and an output-buffer. The input-buffer takes in a signal and sends it out one sample at a time to the adders, delays, and multipliers. The output-buffer collects the resulting samples and combines them into a new signal. While other components transform a signal in one update, this particular process takes several updates of each primitive to completely transform the signal, requiring a "while" loop statement in that subsystem's update algorithm. The "while" loop can be added through AVSI, but the problem occurs when the conditional expression for the loop is built. To tell the subsystem when the last sample of the signal has been sent out, the input-buffer has an output called "done", which is set to true during the update when the last sample has been sent out. At this time, the loop should terminate and a few more updates should be called to the other primitives in the subsystem to process this last sample. However, the "done" output cannot be added to the loop condition in AVSI; it must be entered by editing the application's saved file. This is a problem because it requires the Application Specialist to have knowledge about how Architect stores saved applications, and requires him/her to save, edit, and load the application before he/she can simulate its execution.

5.6.3 Export Values. A related problem to the "state" attributes problem is that export values are undefined at the beginning of the first execution of an application, but on subsequent executions they start out with whatever values they were assigned last during the previous execution. This causes effectively non-deterministic behavior in any application that has a loop (or primitives being updated out of order). The Filter Components primitives example from the last section demonstrates this problem; the adders, delays, and multipliers in the subsystem described above are almost always composed into

a looping structure. An example of this is shown in Figure 5.19. This configuration shows an accumulator in which every sample of the output signal will be the sum of all the previous samples of the input signal. The first time this subsystem is updated, the export value associated with the delay output is undefined, which the adder knows to interpret as a zero. Every time thereafter, even if the delay last-sample attribute is reset to zero, the export value associated with that output will still be set to whatever value was output on the previous update, resulting in the total sum of the previous signal being added in to the current signal. This is not desired. The current work-around for this problem is to reload the application each and every time before execution; when an application is reloaded, all the export values are set to the REFINES “undefined” value (in other words, they are initialized).

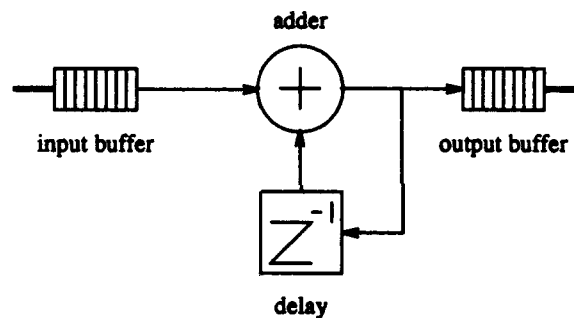


Figure 5.19 An accumulator

5.6.4 Coefficients and Constants. Although coefficients are implemented for primitives in Architect, their purpose is not well-defined and there is no clear method for using them or changing their values. Therefore, we did not implement any coefficients for DSP primitives. Whenever a case came up during the domain implementation phase where we felt that a coefficient would be the logical choice, we used attributes instead. Also, Architect does not have a method for implementing the constants in the OCU model; they must also be implemented as attributes.

5.7 *Summary*

This chapter described the DSP domain implementation accomplished as a part of this thesis effort. This implementation was successful (as demonstrated by the applications shown in Appendix B), and is included in the current version of Architect. We pointed out that McCain's domain analysis approach fits well into our generic knowledge base population method and produces outputs that can easily be entered into Architect's Technology Base. We also identified several cases where the domain analysis was not as independent of the specific DOACS (in this case, Architect) as we had envisioned in our generic population methodology, both because this was a small-scale implementation effort (only one DOACS), as well as the fact that the state-of-the-art for domain analysis is not advanced enough to capture all types of useful information in a standard format. Finally, we discussed four problems in Architect that hinder implementation and use of a domain. These issues will appear in the next chapter as recommendations for future research.

VI. Conclusions and Recommendations

6.1 Results of This Research

From Chapter I (Section 1.2), the problem description for this research was:

Investigate and implement a method to populate the Technology Base (knowledge base) in Architect and demonstrate that a more substantial domain can be implemented in this system.

We were successful in achieving these objectives. First we defined and made generalizations about application composition systems, which resulted in G-DOACS (Figure 3.1). To our knowledge, this generalization had not been attempted before. We then used the G-DOACS concept to help define and generalize about methods to add domain information to DOACSs, which led to the development of our generic knowledge base population methodology (Figure 3.4) ¹. This methodology is mostly a synthesis of three established methods (5, 32, 25), but it does present some new concepts such as keeping the domain analysis phase independent from the requirements of a particular DOACS, incorporating "reusable applications" back into the domain model and domain implementation, and providing a direct and integrated method for evaluation and feedback. We then instantiated this generic methodology for a specific DOACS: Architect (Figure 4.8). This step produced a formalized method to implement domains in Architect, meeting the first objective of our problem statement. To substantiate this method, we analyzed the DSP domain, implemented it in Architect, and validated this implementation by testing the primitives and creating applications whose correct behaviors were known in advance. This process accomplished the second objective in our problem statement.

6.2 Conclusions

Our conclusions concerning the generic knowledge base population methodology we developed are:

¹These first two steps were developed jointly with Sandy.

- Our generic knowledge base population methodology is valid. We instantiated it and used that instantiation to implement the DSP domain. Also, Sandy instantiated the same process for his research in (35).
- Our generic knowledge base population methodology is indeed “generic”. It is not dependent on a particular domain analysis approach nor on a particular DOACS; rather, it was designed to generalize them. We designed this methodology so that it can be instantiated for any current object-oriented domain analysis approach and DOACS, and hope it is able to accommodate future ones.
- Domain analysis should be done as independently from the requirements of a particular DOACS as possible; however, complete independence is not currently possible (as discussed in Section 5.3.1). Our generic population methodology specifies decoupling of the analysis and implementation phases, but the state-of-the-art is not yet advanced enough to completely accomplish this goal, because there are no standardized methods and formats to capture all required domain knowledge. A simple example is adding the domain visualization step to McCain’s domain analysis process (see Section 5.3.2)—as with all current domain analysis approaches, it did not provide for capturing all the information that any DOACS would need. Also, we captured the DSP domain using an architecture very similar to the one used in Architect (the OCU model). This provided a significant time savings since conversion between architectures is not currently well-understood; however, it also constrained the domain analysis process with another Architect requirement. As domain analysis, domain implementation, and application composition in general become more understood, we believe that this division will be a natural result, much like the division between the front and back ends of a compiler (discussed in Section 3.3).
- The domain analysis and domain implementation steps of our process are not one-time affairs. It is impractical to expect that a process as complicated as domain analysis or domain implementation can be done all at once and be consistent and complete. Also, new information (both about the domain as well as domain analysis/implementation methods) will always surface requiring changes; therefore the

domain implementation, as well as the domain analysis results, must evolve over time. Our generic methodology provides for this evolution.

The conclusions we arrived at concerning Architect are

- Architect is capable of composing applications in more sophisticated domains. As stated in Chapter I, before this research effort, only two relatively simple domains had been implemented in Architect. One of the goals of this research was to implement a domain that had more features (more interface types, wider scope, larger number of primitives, etc)—this goal was met by implementing the DSP domain and composing applications in that domain. There are some problems with Architect that limit the DSP domain implementation, however. These problems were discussed in Section 5.6.
- Implementing a domain in Architect is a relatively easy and straightforward task. Additionally, this process is well-defined and is accomplished in the same manner for every domain. If the domain analysis is accomplished correctly, domain implementation is simply an exercise of converting the domain information from the format used in the domain analysis process to the format required by Architect. This type of conversion process lends itself to automation, as discussed in the following section.

6.3 Recommendations for Further Research

Several ideas for further research were identified during our thesis effort. The more important ones are listed below. The first few are more or less independent of Architect; the last few make recommendations about including already research technologies into Architect. Appendix D lists more recommendations for additions to Architect whose scope was not broad enough to be listed here.

- More research must be done in the area of domain analysis/domain implementation. As stated above, this technology is still in its early stages and needs more study before the application composition philosophy can become widespread. Specifically, the type and method of capture of the information that needs to be collected during the domain analysis phase to support the domain implementation in many different systems needs to be better defined.

- Research should be conducted into converting domain information from the knowledge base of one DOACS into the knowledge base of another. This is based on the idea that domain analysis should only be done once for a domain, not once every time that domain is implemented in another DOACS. Based on our research and that of Sandy (35), converting domain information from APTAS to Architect should be possible. One step beyond converting information between DOACSs is to build a system that captures domain information in a DOACS independent form. This system could then be used to supply domain to different DOACSs.
- Methods to validate domains should be studied. In Architect, while the validation of individual primitives is well-defined (using the Test Primitive tool), validating the domain as a whole is not so well-defined. Research in this area may result in a "Test Domain" tool, but as a minimum it should result in a formalized method with clear and concise guidelines.
- Collecting and implementing domain-specific semantic information should be investigated. Architect performs semantic checks on applications but currently only checks constraints from the OCU model. Domain-specific semantic checks should be added; however, how to collect, implement, and use this domain-specific semantic information is not yet well-understood.
- As stated above in our conclusions, domain implementations are not static; they need to evolve. Part of this evolution may include changing the functionality of a primitive. When a primitive changes, however, all the previously created applications need to be changed also. Research needs to be conducted on what to do with previously created applications when a primitive is changed.
- The implementation of domain-specific architectures into Architect should be studied. Currently, Architect has only one architecture (the OCU model). Additional architectures, or the ability of the Software Engineer to create a new architecture for each domain, should be studied for addition to Architect.
- The Architect domain implementation method formalized as a part of this research (Chapter IV) needs to be updated with the database work accomplished by Cecil and

Fullenkamp (7) and the event-driven domain implementation work accomplished by Waggoner (41). As stated in Section 1.3 there were several research efforts involving changes to Architect that were accomplished currently to our research; due to the intractability of attempting to keep all these research efforts up-to-date with each other, they were done independently for the most part². The results of the two research efforts listed above made changes to Architect that affect our research; therefore, when all these independent changes are integrated, our Architect domain implementation method will have to be updated. We say "updated" because the changes should be minor, except for the need to collect timing information during the domain analysis process.

- Additional composition methods should be studied for possible implementation in Architect. Currently Architect has only one composition method: the Application Specialist chooses primitives and connects them together. Another possible composition method might be for the Application Specialist to identify the problem to be solved by specifying parameters and answering questions, then having Architect automatically compose the application. The filter design portion of the DSP domain would be a good validation area for this research.
- Now that this research has formalized the process of domain implementation in Architect, research should be conducted into automating this process. Based on our research effort, we conclude that such a tool could be easily created. For example, the Software Engineer could enter the information required for domain implementation in Architect (see Appendix A) in a high-level form (perhaps by creating a visual object model); this automated tool would transform this information and add it to Architect's Technology Base. Such a tool would save time and reduce errors in the domain implementation process (for example, as it stands now the software engineer must specify the attributes of each primitive three times in three different formats; this takes time and increases the potential for errors).

²As stated in Section 1.3, the one research effort we did completely incorporate into our work was Cossentine's visualization research (9).

After accomplishing this automation, research should be conducted into implementing domains in Architect using automated domain analysis tools (one such tool is OAKS (10), described in Section 3.3). This research should investigate using an automated domain analysis tool to input domain information into Architect's Technology Base directly, as well as tying an automated domain analysis tool to the domain implementation tool discussed above.

While Architect as it currently exists is a fully functional DOACS, there are several features that would add to the system's capabilities. The features identified as a part of this research effort that require further study are listed in Appendix D.

6.4 Concluding Remarks

Current software development methods are not capable of handling current, let alone future, software requirements. Software engineers cannot continue to treat each development effort as a separate and unique problem. Methodologies that focus on automated reuse of domain, software engineering, and problem solving information need to become the rule in software development organizations. Technologies, like application composition, that implement these methodologies will revolutionize the way in which software is created so that development can keep up with demand. This research has contributed to this future of automated, standardized software development by generalizing about application composition systems, creating a methodology to populate applications composition systems, and giving an example of a specific population process for a particular application composition system.

Appendix A. Domain Information Needed for Architect

This appendix summarizes the abstract domain information needed to implement a domain in Architect and the form it takes in that implementation. This information must be contained in the outputs of the domain analysis activity.

1. A Description of the Domain: includes domain assumptions and domain design information, among other things. Basically, include any information that the Software Engineer will need to understand the domain analysis outputs. The Software Engineer will use this description, along with domain implementation information, to develop the domain description that is included with the implementation in Architect.
2. Domain structure: a "tree-like" organization of reusable components (the leaves of the tree – they will become primitives when implemented) and generalization classes (branches in the tree). A name must be assigned to each node in the tree. An example of such a domain structure is shown in Figure 5.17.
3. Interface types: each type of information that will be passed between the reusable components must be defined. For each, a name and description of what data it contains must be defined. An example in the DSP domain is the interface type called complex-signal. The complex-signal type consists of a sequence of pairs of real numbers where each element of the sequence represents one sample, the first real number of each element represents the real part of a complex number, and the second real number of each element represents the imaginary part of the same complex number.
4. Reusable component definitions: for each reusable component listed above, the following must be defined:
 - Description: text describing the purpose and use of this component
 - Attributes: name, type, allowable values, whether or not it should be user-editable, default value, and a text description
 - Coefficients: name, allowable values, and default value

- **Inputs/Outputs:** name and interface type
- **Icon(s):** for Architect, these icons must be bitmaps
- **Update function:** for implementation in Architect, this function must be specified in **REFINE**

This short list is all the information needed in the domain analysis outputs to implement a domain in Architect.

Appendix B. Digital Signal Processing Examples in Architect

This appendix will consist of a two examples of DSP applications. The first is a conical DSP application called a moving average, shown in Figure B.1. This particular application is a four-sum moving average: each sample of the output is the average of the current output along with the three previous samples. The multiplier primitive (looks like a triangle) has a value of 0.25 for its multiply value. The input signal loaded from a file was generated by a previous application that simply added a sinusoid with some noise. Note that this application displays two signals: the original input (shown in Figure B.2) and the output after going through the four-sum moving average part (shown in Figure B.3).

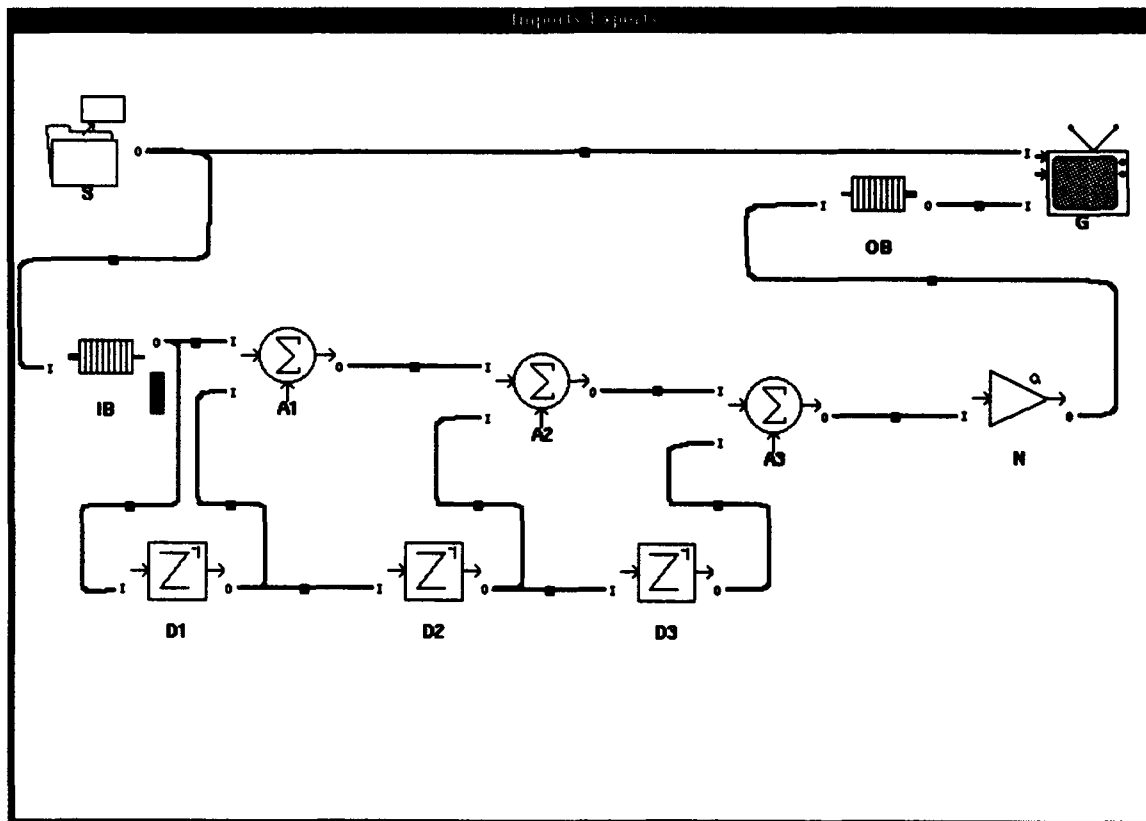


Figure B.1 A Four-Sum Moving Average

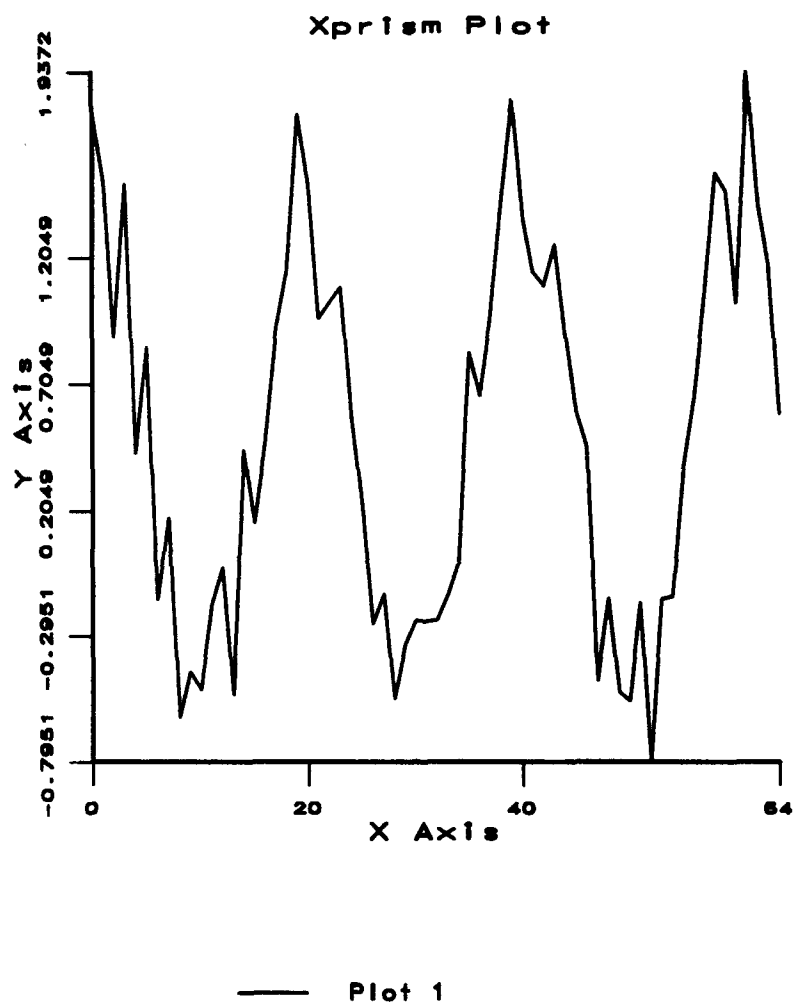


Figure B.2 The Input to the Four-Sum Moving Average

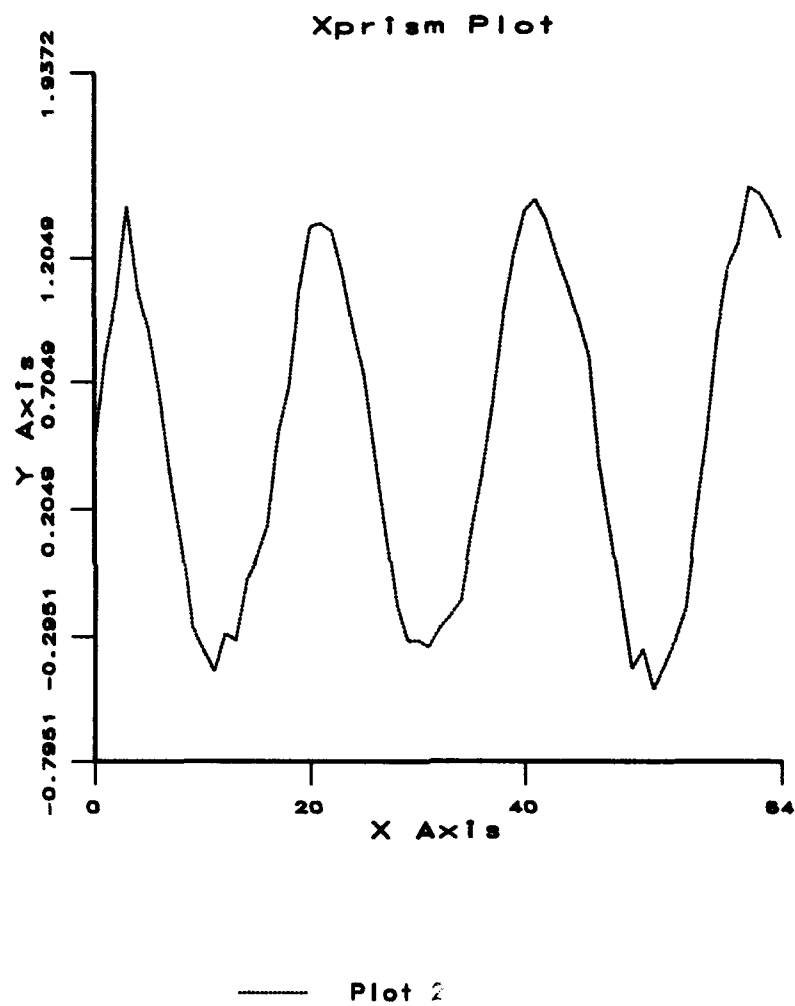


Figure B.3 The Output from the Four-Sum Moving Average

The second application, called Window-Demo, shows why windowing should be used before a Fourier transform is applied. For this application, shown in Figure B.4, a sinusoid is transformed in two different ways and then both are displayed. The lower path takes the input sinusoid through a Fourier transform and a complex-to-real conversion (the DFT primitive outputs a complex signal, but the graph2-signal primitive can only show real signals); the output from this path is shown in Figure B.5. The upper path takes the input sinusoid through a window, and then through a Fourier transform and a complex-to-real conversion just like the lower path; the output from this path is shown in Figure B.6. Note that this output is much closer to the canonical representation of a sinusoid after a Fourier transform has been applied, which is the purpose of windowing a signal.

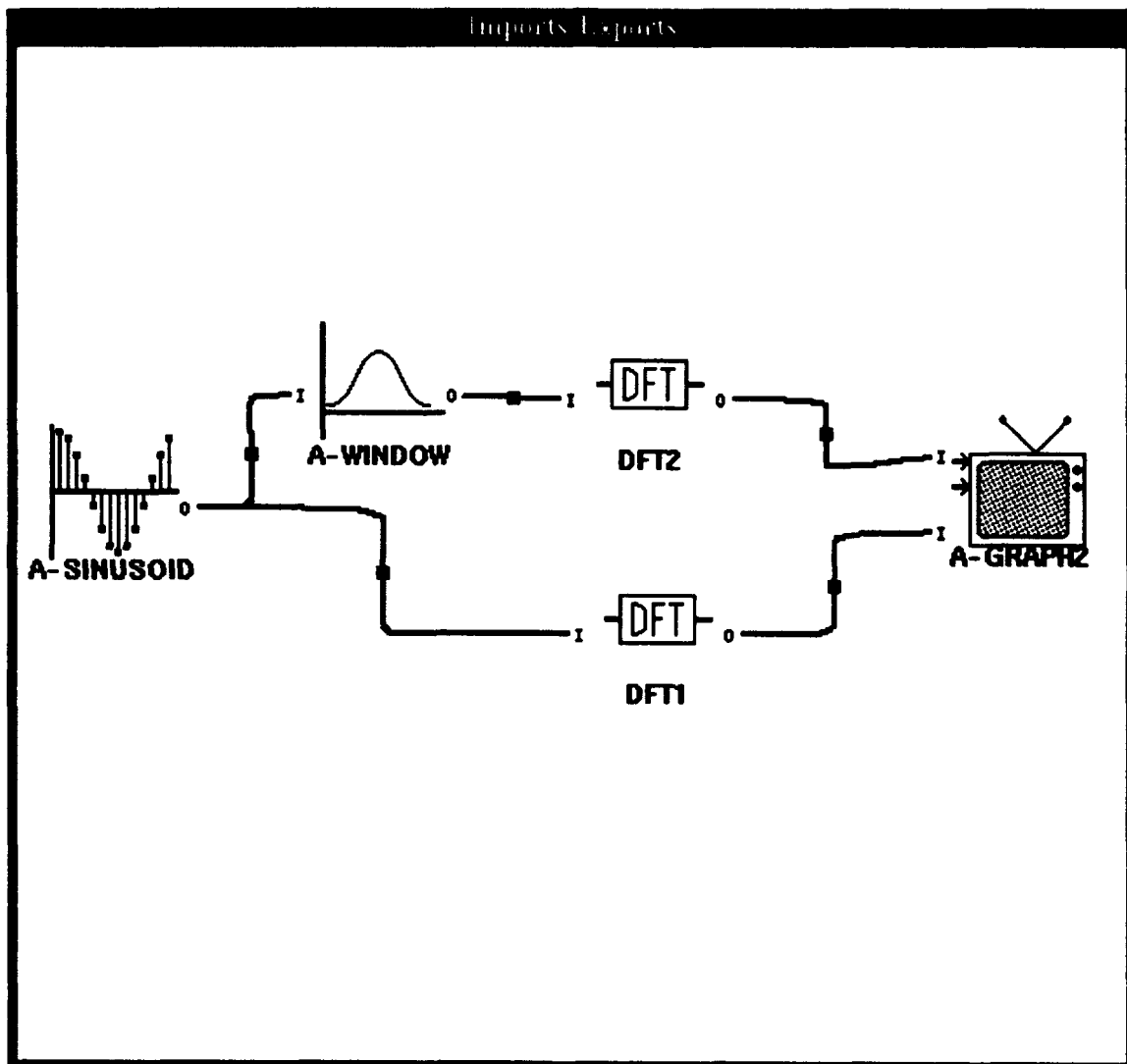


Figure B.4 The Window-Demo Application

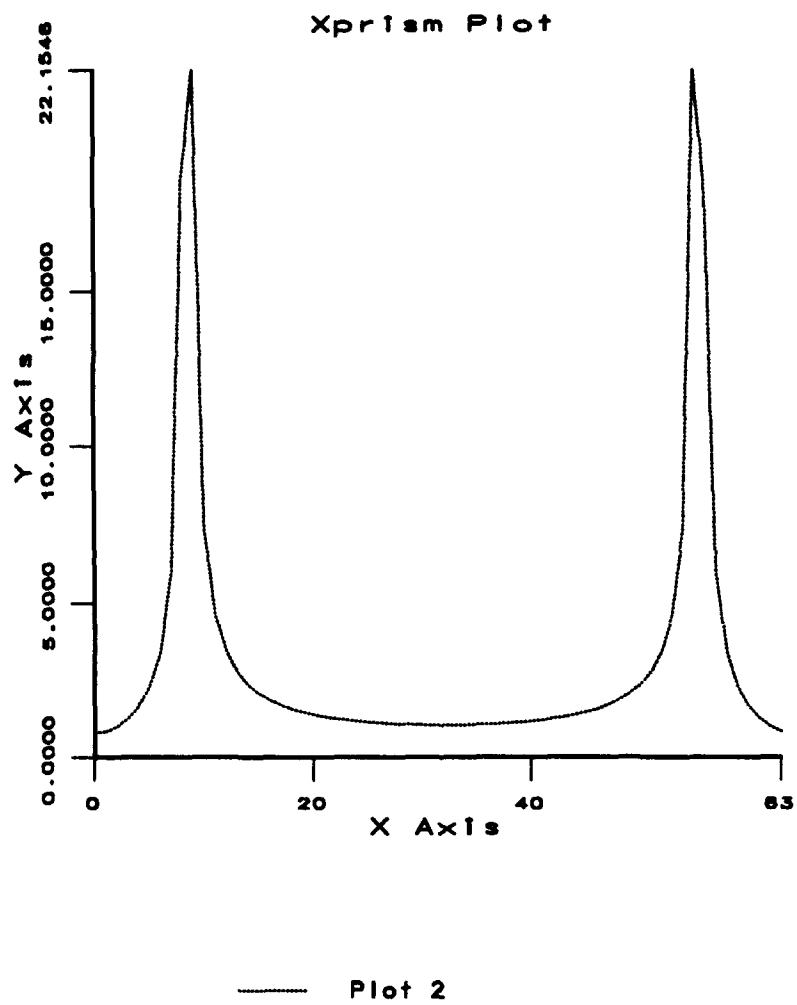


Figure B.5 The Fourier Transform without Windowing

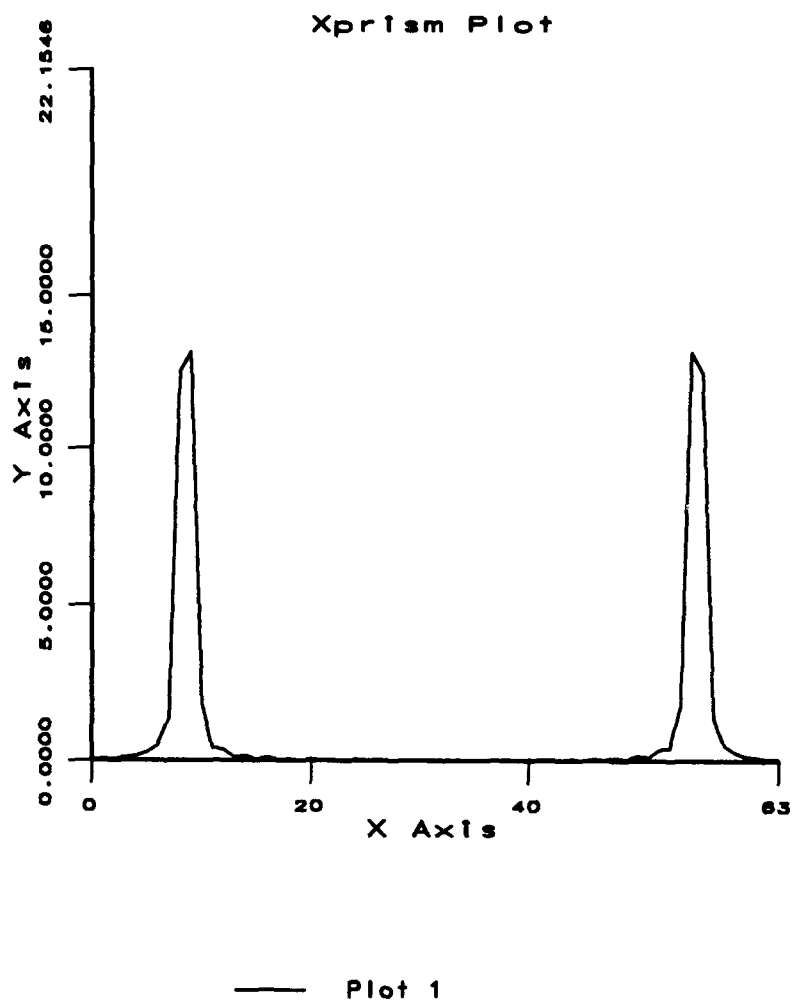


Figure B.6 The Fourier Transform with Windowing

Appendix C. File Conventions for Architect

This appendix contains the file conventions currently being used for Architect.

Although all the code to implement a domain in the Technology Base could be entered into one file, this file would be confusing, making changes difficult. To alleviate this problem, file conventions are being used to manage this information. The following files contain the domain information for Architect's Technology Base:

1. `dm-<domain-name>.re`: this file contains the domain-specific types, domain-specific functions, domain class declarations (with structure), and the domain description. This file includes the code in Figures 5.6 and 5.7.
2. `<primitive-name>.re`: one file for each primitive. This file contains the primitive inputs and outputs, attributes, coefficients, description, and update function. This file includes the code in Figures 5.8 and 5.15 for the DFT primitive.
3. `<primitive-name>.icon-l` and `<primitive-name>.icon-s`: one each for every primitive. These files contain the bitmaps that are used to create the primitive icons in Architect. Currently, these files are created using the OpenWindows IconEdit tool.
4. `gram-<domain-name>.re`: this file contains the DSL grammar. A production is created for each primitive as shown in Figure 5.9 for the DFT primitive. All attributes for a primitive are listed except for "state" attributes that should be initialized to their default values (specified in the previous file) every time the a primitive of that class is loaded with an application.
5. `vsl-<domain-name>.re`: this file contains the domain visual specification that is parsed in through Architect's VSL grammar. A section is entered for each primitive; in this section, the attributes that should be editable by the Application Specialist are entered (the un-editable attributes are not entered). This file includes the code in Figure 5.10 for the DFT primitive.
6. `var-<domain-name>.re`: this file contains two lines for each primitive that simply create the icon objects for each primitive. This file includes the code in Figure 5.10 for the DFT and Sinusoid primitives.

The above file conventions are just one of a possible set of ways to organize the code to implement a domain in `REFINE`. Comparing these file descriptions with the information in the previous appendix, it can be seen that the first three types of files contain the bulk of the information, the rest mostly repeat the information defined in these first three.

Appendix D. Future Recommended Features for Architect

This appendix summarizes the features identified during this research effort that we feel should be studied for incorporation into Architect.

1. There are currently no bounds on the value that can be assigned to primitive attributes other than the standard type checking. It would be useful if the Software Engineer could add additional bounds on the allowable values of an attribute; for example, a primitive may have two attributes such that the value of one must be less than the value of the other. The implementation of further restrictions should be studied.
2. Although user-defined types (like real-signal-type) can be used to define primitive attributes and input/outputs, they cannot be accessed in the if/while statement conditional part. Currently, only real, integer, string, and boolean variables are allowed in these statements. The ability to access user-defined types in these statements should be added.
3. Integration of domain-independent primitives into Architect should be studied. For example, the two DSP primitives that convert real numbers to/from complex numbers should not be DSP-specific; they should be provided to any domain.
4. The ability to take a subsystem and add it to the technology base window with the primitives should be added. This would allow the Application Specialist to add the functionality of such a subsystem to applications without recreating and revalidating it every time.
5. OCU component attributes, constants, and current_state variables are all implemented in Architect as attributes; this causes problems in the DSP domain (discussed in Section 5.6). These three variables should be implemented differently from each other, or a method to distinguish between them should be added.
6. Several of the DSP primitives implemented as a part of this research have the same functionality, the only difference is that they have a different number of inputs. This is because all inputs must be connected before applications can be executed;

however some components defined in the domain analysis do not have a fixed number of inputs. One example in the DSP domain is the four graph-X-signal, where X is 1, 2, 3, or 4. If Architect had the ability to have optional inputs on primitives, then only one graph primitive would have had to be implemented. Also, the ability to have N-input primitives (where the Application Specialist specifies N) would be useful. An example in the DSP domain where this would be useful is the adderX primitives; only three of them were implemented because there are no bounds on X for this primitive.

7. Some attribute types lend themselves to enumerated types; however, Architect does not have the capability to handle these enumerated types. An example in the DSP domain where this would be useful is the real-to-complex primitive. This primitive has an attribute called conversion-type which tells the primitive what method to use in converting the signal; there are four types: real, imaginary, magnitude, and phase. When we implemented this primitive, we had to implement it as a symbol and indicate in the attribute description that there were only four allowable values; an enumerated type would have been much better
8. When making primitive connections internal to a subsystem, if an output is connected to an input in that subsystem, there is nothing to show whether or not it is also connected to another input outside the subsystem. This problem hides important information from the Application Specialist.
9. Architect has the ability to use generically specified applications (called generics) as described in (3) and (33); however, their purpose is not well-understood and AVSI does not support them. Hence, further research would be valuable.

Appendix E. REFINE Code Listings for Architect

The REFINE source code for Architect and the implemented DSP domain may be obtained, upon request, from:

Maj Paul Bailor
AFIT/ENG
2950 P Street
Wright-Patterson AFB, OH 45433-7765

(513)255-9263
DSN 785-9263
email: pbailor@aft.af.mil

Bibliography

1. Aho, Alfred B., et al. *Compilers: Principles, Techniques, and Tools*. Mark S. Dalton, 1985.
2. Alkin, Oktay. *PC-DSP*. Prentice-Hall, Inc, 1990.
3. Anderson, Cynthia. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/92D-01, Graduate School of Engineering, Air Force Institute of Technology (AU), December 1992.
4. Arango, Guillermo. "Domain Analysis - From Art Form to Engineering Discipline." *Proceedings of the Fifth International Workshop on Software Specification and Design*. 152-159. May 1989.
5. Arango, Guillermo. "Domain Analysis, From Art Form to Engineering Discipline." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 81-88, IEEE Computer Society Press, 1991.
6. ASD/RWW. *Software Structural Model (SSM) Design Methodology for the Modeling Library Components for the Joint Modeling & Simulation System (J-MASS) Program; Version 1.1*. Technical Report, Architectural Technical Working Group, April 1992.
7. Cecil, Danny A. and Joe Fullenkamp. *Using Database Technology to Support Domain-Oriented Application Composition Systems*. MS thesis, AFIT/GCS/ENG/93D-03, Graduate School of Engineering, Air Force Institute of Technology (AU), December 1993.
8. Coglianese, Lou and Will Tracz, "NAECON '93 DSSA-ADAGE Workshop and Tutorial." workshop handout, May 1993.
9. Cossentine, Jay A. *Developing a Sophisticated User Interface to Support Domain-Oriented Application Composition and Generation Systems*. MS thesis, AFIT/GCS/ENG/93D-04, Graduate School of Engineering, Air Force Institute of Technology (AU), December 1993.
10. Crowley, Nancy. *On the Automation of Object-Oriented Requirements Analysis*. PhD dissertation, Department of Engineering, Air Force Institute of Technology, September 1993.
11. Elliott, Douglas F., editor. *Handbook of Digital Signal Processing: Engineering Applications*. Academic Press, Inc., 1987.
12. Gool, Warren E. *Alternative Architectures for Domain-Oriented Application Composition and Generation Systems*. MS thesis, AFIT/GCS/ENG/93D-11, Graduate School of Engineering, Air Force Institute of Technology (AU), December 1993.
13. Hayes-Roth, Frederick, et al. *Building Expert Systems*. Addison-Wesley Publishing Company, Inc., 1983.

14. Iscoe, Neil. "Domain-Specific Reuse: An Object-Oriented and Knowledge-Based Approach." *Tutorial on Software Reuse: Emerging Technology* edited by Will Tracz, IEEE Computer Society Press, 1988.
15. Iscoe, Neil. "Domain-Specific Reuse: An Object-Oriented and Knowledge-Based Approach." *Tutorial from Software Reuse: Emerging Technology* edited by Will Tracz, 299-308, IEEE Computer Society Press, 1989.
16. James Rumbaugh, et. al. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Pentice Hall, Inc, 1991.
17. Jensen, Paul S. and Lori Ogata. *DRAFT Final Report for Automatic Programming Technologies for Avionics Software (APTAS)*. Technical Report, Lockheed Software Technology Center, July 1991.
18. Kang, Kyo C. and others. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report, Software Engineering Institute, Carnegie Mellon University, November 1990.
19. Kuc, Roman. *Introduction to Digital Signal Processing*. McGraw-Hill, Inc., 1988.
20. Lee, Kenneth J. and others. *Model-Based Software Development (draft)*. Technical Report, Software Engineering Institute, December 1991.
21. Little, John N. and Loren Shure. *Signal Processing TOOLBOX: User's Guide (For Use with MATLAB)*. MathWorks, 1992.
22. Lowry, Michael R. "Software Engineering in the Twenty-first Century." *Automating Software Design*, edited by Michael R. Lowry and Robert D. McCartney. 627-654. Menlo Park, CA: AAAI Press, 1991.
23. Lubars, Nitchell D. "Domain Analysis and Domain Engineering in IDEa." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 163-177, IEEE Computer Society Press, 1991.
24. Ludeman, Lonnie C. *Findamentals of Digital Signal Processing*. Harper & Row, Publishers, Inc., 1986.
25. McCain, Ron. "Reusable Software Component Construction: A Product-Oriented Paradigm." *AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference*. 125-135. AIAA, October 1985.
26. Neighbors, James M. *Software Construction Using Components*. PhD dissertation, University of California, Irvine, 1981.
27. Neighbors, James M. "Draco: A Method for Engineering Reusable Software Systems." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 34-52, IEEE Computer Society Press, 1991.
28. Ogush, Mike. "A Software Reuse Lexicon," *CrossTalk* (December 1992).
29. Oppenheim, Alan V. and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice-Hall, Inc., 1989.
30. Parks, T. W. and C. S. Burrus. *Digital Filter Design*. John Wiley & Sons, Inc., 1987.

31. Prieto-Díaz, Rubén. "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, 15:47-54 (April 1990).
32. Prieto-Díaz, Rubén. "Domain Analysis for Reusability." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 63-69, IEEE Computer Society Press, 1991.
33. Randour, Marry Anne. *Creating and Manipulating a Domain Specific Formal Object Base*. MS thesis, AFIT/GCS/ENG/92D-13, Graduate School of Engineering, Air Force Institute of Technology (AU), December 1992.
34. Reasoning Systems, Inc. *REFINE User's Guide*. 3260 Hillview Ave., Palo Alto, CA 94304, May 1990.
35. Sandy, Raleigh A. *A Method for Populating the Knowledge Base of APTAS, a Domain-Oriented Application Composition System*. MS thesis, AFIT/GCE/ENG/93D-13, Department of Engineering, Air Force Institute of Technology, December 1993.
36. Smith, Douglas R. "KIDS: A Semiautomatic Program Development System," *IEEE Transactions on Software Engineering*, 16:1024-1043 (September 1990).
37. Stearns, Samuel D. and Ruth A. David. *Signal Processing Algorithms Using Fortran and C*. P T R Prentice-Hall, Inc., 1993.
38. Stearns, Samuel D. and Ruth A. David. *Signal Processing Algorithms Using Fortran and C*. P T R Prentice-Hall, Inc., 1993.
39. Tracz, Will. "Domain Analysis Working Group Report - First International Workshop on Software Reusability," *ACM SIGSOFT Software Engineering Notes*, 17:27-34 (July 1992).
40. University of New Mexico. *Visual Programming System and Software Development Environment for Data Processing and Visualization (Khoros)*, 1991. Release notes for Khoros release 1.1.
41. Waggoner, Robert. *Implementing Time-Dependent Technology Bases in Architect*. MS thesis, AFIT/GCS/ENG/93D-23, Graduate School of Engineering, Air Force Institute of Technology (AU), December 1993.
42. Warner, Russell M. *A Method for Populating the Knowledge Base of AFIT's Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93D-24, Graduate School of Engineering, Air Force Institute of Technology (AU), 1992.
43. Wartik, Steven and Rubén Prieto-Díaz. "Criteria for Comparing Reuse-Oriented Domain Analysis Approaches," *International Journal of Software Engineering and Knowledge Engineering*, 2:403-431 (September 1992).
44. Weide, Timothy. *Development of a Visual System Interface to Support a Domain-oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93M-05, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, March 1993.

45. Welgan, Robert L. *Domain Analysis and Modeling of a Model-Based Software Executive*. MS thesis, AFIT/GCS/ENG/93D-25, Graduate School of Engineering, Air Force Institute of Technology (AU), December 1993.

Vita

Captain Russell M. Warner was born September 25, 1964 in Enid, Oklahoma and graduated from Caesar Rodney High School in Dover, Delaware in 1983. He attended the University of Delaware for a year before being accepted to the United States Air Force Academy in 1984. After graduating with honors in 1988, Russell was commissioned a Second Lieutenant in the Air Force and assigned to the Directorate of Communications/Computer Systems, Headquarters, Electronic Security Command (later renamed Air Force Intelligence Command). He married Marilee S. Laursen, also a graduate of the Air Force Academy, on 2 June, 1988. In May, 1992, he entered the Air Force Institute of Technology to pursue a Master of Science degree in Computer Engineering.

Permanent address: 619 Carriage Ln.
Dover, DE 19901

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE December 1993		3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE A METHOD FOR POPULATING THE KNOWLEDGE BASE OF AFIT'S DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM				5. FUNDING NUMBERS
6. AUTHOR(S) Russell M. Warner				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/93D-24
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Capt Rick Painter 2241 Avionics Circle, Suite 16 WL/AAWA-1 BLD 620 Wright-Patterson AFB, OH 45433-7765				10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Distribution Unlimited				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 words) This research developed a formal method for adding new domains to Architect, a domain-oriented application composition system being developed at the Air Force Institute of Technology (AFIT) to explore new software engineering technologies. Using canonical formal specifications of domain objects, Architect rapidly composes these specifications into a software application and executes a prototype of that application as a means to demonstrate its correctness before any programming language specific code is generated. Architect is implemented in the Software Refinery environment, which allows Architect to create and manipulate object-oriented specifications. As a part of this research effort, domain-oriented application composition systems were investigated in general, leading to the development of a general method for populating the knowledge base of systems of this type. This general population method was then used as a basis for creating a specific knowledge base population method for Architect. To validate this method, Architect was populated with the Digital Signal Processing domain. The correct implementation of this domain was verified by creating applications and comparing their execution to expected results. The addition of the Digital Signal Processing domain to Architect also serves to validate the usefulness and correctness of the Architect system.				
14. SUBJECT TERMS Software Engineering, Automatic Programming, Knowledge Based Systems, Domain Modeling, Domain-Specific Languages, Application Composition Systems				15. NUMBER OF PAGES 130
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	